HIGHLY SCALABLE DATA BALANCED DISTRIBUTED SEARCH STRUCTURES

By

PADMASHREE KRISHNA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
UNIVERSITY OF FLORIDA

1995

To my parents Mr Apparao and Mrs Meenakshi my sisters Lakshmi and Kutty
my brother Ravi, my husband Krishna and lastly my son, Ankith

ACKNOWLEDGEMENTS

I want to thank my Ph.D committee chairman Dr Theodore Johnson, for the uncountable and invaluable number of hours he has spent in guiding me through this research There were times when I have come out of his office more confused than when I went in but that set me thinking along the right direction I thank him for his invaluable suggestions and critical remarks that have led to this research

I wish to thank my other committee members for, firstly, agreeing to be on my committee and then offering helpful suggestions along the way

I thank Dr Randy Chow and Dr Haniph Latchman my external committee member, for showing a lot of interest in my research and for the useful suggestions they provided along the way

Dr Sartaj Sahni offered critical remarks on the research, which led me to rethink some ideas and design better experiments Dr Sahni's main concern was the scalability of the B-trees with large degrees and with a large number of keys

Lastly, Dr Richard Newman Wolfe deserves special thanks for introducing me to my advisor in the first place, providing helpful hints on my research and on writing the dissertation

I would like to thank all the CIS staff, in particular Mr John Bowers the graduate secretary, for their help

TABLE OF CONTENTS

CHAPTERS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

HIGHLY SCALABLE DATA BALANCED DISTRIBUTED SEARCH STRUCTURES

By

Padmashree Krishna

May 1995

Chairman: Theodore J. Johnson
Major Department: Computer and Information Sciences

Present trends in parallel processing and distributed databases necessitate the maintenance of large volumes of data. Scalable distributed search structures provide the necessary support for mass storage. In this research, we focus on the performance of two large scale data-balanced distributed search structures, the dB-tree and the dE-tree. The dB-tree is a distributed B-tree that replicates its interior nodes. The dE-tree is a dB-tree in which leaf nodes represent key ranges, and thus requires far fewer nodes to represent a distributed index.

The main objective is to develop distributed algorithms and protocols and implement them to study their performance. The first concern is the basic distribution of the data structure. Distributed storage in turn calls for data balancing to utilize the system resources efficiently and to avoid overloading any single processor. The advantage of distributed storage would be lost if there was no replication. Replication of the B-tree index nodes is necessary to avoid the root bottleneck and enhance parallelism.

The algorithms for data balancing determine how the tree nodes are assigned to processors. Here, we develop several algorithms for data balancing, both for the dB tree and the dE-tree. We find that a simple distributed data balancing algorithm works well for the dB-tree, requiring only a small space and message passing overhead. We compare three algorithms for data balancing in a dE-tree, and find that the most aggressive of the algorithms makes the dE-tree scalable.

We have developed two algorithms for replication, namely full replication and path replication and studied their performance. We have observed that path replication performs better and permits algorithms to scale to large trees.

We have also performed some timing experiments on our dB tree to study the response times and throughput of our system. The experiment was performed on 4, 6 and 8 processors. To provide an explanation of the response times obtained, we performed experiments to obtain the message transmission time and processing time.

We have developed an analytical performance model of the dB-tree and the dE-tree using the data from the simulation experiments. We then applied the model to our experimental parameters to obtain the predicted response time and determined that our analytical model predicts a more pessimistic timing than the timing we obtained from our experiments. Our experiments give a value of 50 milliseconds response time with 8 processors, whereas the model predicts 56.5 milliseconds. From the analytical model, we observe that a distributed search structure permits a much larger throughput than a centralized index server, at the cost of a modestly increased response time.

CHAPTER 1
INTRODUCTION

1.1 Objective

The main objective of this research was to develop distributed algorithms and protocols for some specific data structures, and implement them to study their feasibility and performance. We approached the problem in two dimensions

1. Algorithmic The algorithmic approach attempts to design distributed algorithms for specific purposes and to study their correctness criteria. Distributed data structures are useful in designing general purpose distributed algorithms However not all algorithms designed can be implemented efficiently We intended to study the implementation of dynamic algorithms on a network of processors without shared memory

2. Implementation. From the implementation view point, we were concerned with the efficiency and performance of the algorithms The implementation of these distributed data structures will hide from the application's user, the details of the sites where data are stored the access methods and the synchronization techniques

For the purposes of the research we selected the B-tree for its flexibility and its practical use in indexing large amounts of data.

1.1.1 Why Distributed Search Structures Were Chosen

Current commercial and scientific database systems deal with vast amounts of data. Since the volume of data to be handled is so large, it may not be possible to store all the data in one place. Also, when addressing large volumes of data, there is the danger of memory bottlenecks. Therefore, distributed techniques are necessary to create large-scale, efficient, distributed storage [24]. Distributed data structures allow for large amounts of data to be manipulated. The data can be stored by partitioning them among the storage sites of the system, which also allows for parallel access to the data. Distributed data structures are useful for many distributed applications (e.g., in permanent information storage and retrieval techniques, global name servers in networks, resource allocation, etc.). Although a considerable amount of research has been done in developing parallel search structures on shared memory multiprocessors, little has been done on the development of search structures for distributed memory systems. One such search structure is the B tree. The B-tree was selected because of its flexibility and its practical use in indexing large amounts of data.

A *distributed system* is a collection of processor-memory pairs connected by an interconnection network. Distributed systems have several advantages over centralized systems because they enable ease of expansion, provide increased reliability, allow actual geographic distribution, and have a higher potential for fault-tolerance and performance due to the multiplicity of resources. Each processor-memory pair will henceforth be called a *site*. Sites communicate by message passing. It is believed that message passing multiprocessors are highly scalable. In a distributed system no single site has complete, accurate and up-to-date information of the global state of the system. Thus, each site must have the capability of handling inaccurate and out-of-date information. Distributed algorithms must tolerate these inconsistencies

1.1.2 The Need for Distributed Data Structures

The data structures used in an algorithm have a considerable effect on the efficiency of the algorithm. Hence, for distributed algorithms, there is a need for distributing the data structures as follows:

1. The primary reason for distributed data structures is that in a distributed system we wish to share the data between processes on different processors. The various parts of the distributed system share data by communication. Several programming languages only support shared variables that allow for pseudo-parallelism of the processes running on the same processor. Simple shared variables can be implemented by simulating shared physical memory, but this is not sufficient for distributed systems that call for complex data structures. Instead, there are basically three ways of providing the notion of shared data to a distributed system:

   (a) Distributed data structures

   (b) Shared logical variables

   (c) Distributed objects in distributed shared memory

2. A secondary reason for distributed data structures is the problem of maintaining large data structures at one physical location. This not only requires a large amount of memory but also makes the system less fault tolerant. Distributing a data structure over a large number of processors implies partitioning the data structure into parts that are individually managed by a single processor. The parts may be disjointed or they may be replicated to provide better

locality and increase availability But in replication there is the problem of inconsistency Distributing the shared data over the different processors improves performance since data residing at different sites can be accessed in parallel

Several programming languages provide the above notion of shared data where the user is unaware of the physical distribution of the data.

1.1.3 The Principle of Data Distribution

Data organisation must be based on the *principle of ubiquity* defined as

- all data objects are accessible to all sites;
- on an access the most recent version of the data is provided;
- consistency is maintained on a global basis

To achieve these criteria, data must be replicated and updates must be atomic All these criteria improve the performance and reduce the cost of access by allocating data depending on the locality of a process In some data structures, the access nature is predictable, while in others not so

Data structures are characterized by the operations they support A distributed data structure consists of a set of local data structures storing the data at various sites of the system and a set of protocols for access to the distributed data structures These protocols specify the query and update operations on the datum The distribution of the data structure is known as the data organisation scheme [2] and may be based on several criteria as follows

1 To improve the locality of the process running on the processor

2 To reduce the message complexity of access to remote data

3. To balance the data across processors for efficient usage

4. To improve the fault tolerance and increase availability

5. To minimize the delay performance per access

Excessive communication among processors can offset the advantage of distributed systems. A good strategy would be to take into account the computation and communication cost imposed by the underlying machine architecture. Several strategies have been proposed for efficient allocation of data structures [9]. The access protocols specify the primitive operations that are to be performed on the data structures and the mode of access by processors to the data.

1.1.4 The Need for Replication

Redundancy or replication is an inherent part of the design of distributed data structures. Not only does replication provide fault tolerance in the event of the failure of a processor, but it also enables dynamic data balancing and reduces costs by placing the more often accessed data close to the processor. A process can take advantage of its locality to reduce the cost of communication. Replication also increases the availability of data. A factor that has to be considered is the degree of replication, also known as replication control. In what is called the *total* structure, all the data are replicated at each processor [46]. This increases the availability and fault tolerance but places a high demand on memory requirements. A compromise is to set up a balance between memory usage and cost considerations [2]. Replication introduces the problem of maintaining consistency among the various copies of the data structure.

1.1.5 Distributed Data Structure Issues

Distributing data structures creates new issues not present in a shared memory or a single processor system. Two basic problems are those created by the concurrency of automatic data partitioning operations and those introduced due to the distribution of the data.

Concurrency issues are resolved by imposing the serializability criteria. Various serializability criteria have been studied in connection to databases [8].

The study of the distribution of data structures and the relationship to the underlying system of processors may lead to efficient schemes for distributing the data in terms of space, time and message complexity [2]. The complexity of data movements is also an issue for distributed structures.

The search structure selected for this research is the B tree. We address all the above issues with respect to distributing a B tree. We have selected the B-tree because of its flexibility and its practical use in indexing large amounts of data.

1.2 Background

1.2.1. Introduction

In this section, we present a survey on the research done in distributed data structures. Techniques that some programming languages provide to support distributed data structures are presented. A brief discussion of basic distributed data structures is then presented. In the discussion of search structures that this research concentrates on, we focus on hash tables, dictionaries and concurrent B trees. Some background on data balancing is also presented. Finally, the concurrent B tree link algorithm is presented, which forms the basis for the distributed B tree algorithms.

1.2.2 Programming Language Support for Distributed Data Structures

Distributed memory machines are much more difficult to devise algorithms for than the shared memory machines. This is due to the lack of a single global address space. The programmer is responsible for distributing code and data to different sites and managing communication between processes. This may reduce programmer productivity, therefore programming languages need to provide facility for developing parallel and distributed programs. In the current conventional programming languages, each process can only access its local address space, which results in large data structures that must be partitioned across the processors. Since interprocessor communication is usually more expensive than computation, it is essential that much of the computation be done using local data. Several programming languages are being developed to support distributed data structures. Some examples are Linda [1], [10], Orca [2], [4] and Kali [32]. Some programming languages provide distributed data structures explicitly, while others do so implicitly. Examine the following three.

1. **Linda**

   The distributed data structure paradigm was first introduced in the language Linda, implemented on AT&T Bell Lab's Net multicomputer [1]. The Tuple Space concept is used for implementing distributed data structures. This tuple space, consisting of tuples that are in an ordered sequence of values, forms a global memory shared by all the processes in the system. To modify a tuple, a "read, modify and write" atomic operation is needed. If two processes want access to a tuple, only one of them succeeds while the other blocks. A distributed array is implemented as a tuple consisting of $< arrayname, index, value >$. The tuples are distributed across processors based on the following criteria.

   (a) Either the entire tuple space is replicated, or

(b) The last processor to create a tuple is the owner of the tuple, or,

(c) A hashing function is used to distribute the tuples.

Communication through distributed data structures is anonymous (as opposed to interprocess communication). Communication primitives such as message passing and remote procedure calls are simulated using the tuple space. The processes interact only through the tuple space. The goal of Linda is to relieve the programmer from the task of parallel programming.

2 Orca

This programming language is mainly intended for developing parallel algorithms for distributed systems. The data structures are encapsulated into passive objects and can be shared by different processors. The objects are replicated on all processors and are updated by a reliable ordered broadcast primitive [2], [4].

3 Kali

A programming environment, Kali is designed to aid in the programming of distributed memory architectures [32]. It allows the programmer to treat the distributed data structures as single objects. A software layer supports a global name space. Algorithms can be specified at a high level and the compiler transforms the high level specification into a set of tasks that interact by "message passing." Thus, the programmer is relieved of the task of programming with low level message passing primitives and can concentrate on pure algorithm development. The only data type supported in this is distributed arrays.

1.2.3 Distributed Data Structures

Here, we present mechanisms of how some of the basic data structures have been distributed. Scalar variables are usually replicated on each processor.

Arrays:

Presently, only distributed arrays are predefined by Kali. However, Kali supports user defined distributions. Array distributions are specified by a distribution clause [32]. The clause specifies a set of distribution patterns for each dimension of the array. An asterisk in the dimension indicates no distribution. The number of array dimensions that are distributed cannot exceed the number of processors in the system. Each processor stores a single copy of each array element.

Another automated data partitioning scheme has been proposed for distributed arrays [15]. This is a constraint-based approach, wherein the compiler analyzes each loop and, based on performance considerations, identifies some constraints on the distribution of data structures. Finally, the compiler tries to combine constraints for each data structure so that the overall execution time of the program is minimized. The data may have to be repartitioned between program segments and between procedure calls. This has been implemented on the Intel IPSC/2 hypercube.

Queues:

A queue is a First In First Out (FIFO) structure that has two ends: a front and a rear. A queue can be stored in a distributed system by storing different segments of the queue in different sites, with each queue element being stored in exactly one site.

Lee et al. have presented a scheme for a fault tolerant distributed queue to provide a high degree of availability, greater flexibility and low access cost [36]. In this scheme s replicas of the queue are made and each replica is broken into s not necessarily in equal-sized segments. Each site maintains the front and rear of a segment of a replica. There is no concurrency implemented as only one site in the entire system is allowed to perform insertion or deletion at a time.

Priority queues have been implemented using systolic arrays. Systolic search trees have also been used to implement multiqueues [38].

1.2.4 Search Structures

Efficient search structures are needed for maintaining files and indices in conventional systems which have a small primary memory and a larger secondary memory. To access individual entities of a file, an index is required. The normal operations carried out on an index are search, insert, and delete. A search table is a data structure in which records are organized in a well-defined manner. Search structures are used for the implementation of dictionaries. An implementation of a search table could be designed using either a tree, an array or a hash table on a sequential machine.

In a traditional design, an access takes a long time to complete, usually on the order of the number of elements stored. In sequential systems data structures such as trees, sorted arrays and hash tables have been used to implement search tables. Of these, the hash table gives the best performance, with little space overhead. Parallelism is achieved by pipelining the accesses, however the sequential nature of the accesses creates a bottleneck. Therefore, a simultaneous design that accepts and handles consecutive accesses concurrently is necessary.

Distributed memory data structures have been proposed by Ellis [14], Severance [54], Peleg [46], Colbrook et al. [12] and Johnson and Colbrook [23]. Colbrook et al. [12] have proposed a pipelined distributed B tree, where each level of the tree is maintained by a different processor. The parallelism achieved is limited by the height of the B tree and the processors are not data balanced. Parallel B trees using multi version memory have been proposed by Wang and Weihl [60] The algorithm uses a special form of software-controlled cache coherence.

Hash Tables

Hashing is a well known technique for fast access to records in a large database. One of the main goals is to provide fast concurrent access Several methods of hashing have been proposed which include distributed linear hashing [54 11], extendible hashing [14], two-phase hashing [61], trie hashing [40] and linear hashing for distributed files [41]

a Distributed Linear Hashing

In linear hashing the table is gradually expanded by splitting the buckets until the table has doubled its size Splitting means rehashing of a bucket **b** and its overflows in order to distribute the keys in them between **b** and one other location Linear hashing requires the use of a series of hashing functions, a new one arising everytime the hash table is doubled

A distributed linear hashing method particularly useful for main memory databases has been discussed by Severance and Pramanik ([54]) In linear hashing, the records are distributed into buckets that are stored on disk, but in distributed linear hashing the buckets are stored in main memory First a bucket is located, and second, the record chain in the bucket is found by another computation By having pointers in them, the records in the bucket can be placed in any memory module An index is used to point to the bucket directories and is cached in each processor Address computation is done locally To avoid hot spots in accessing the central variables, a local copy is kept at each processor The local copies may be out-of date at times causing incorrect bucket address computation Retry logic is used to solve this problem The paper also addresses the problem of maintaining local copies of the centralised variables and recovery mechanisms The design has been implemented on a BBN Butterfly multiprocessor system

- **Extendible Hashing**

Extendible hashing combines radix search trees (or tries) and hashing. To represent the index, an extendible trie structure is used instead of a binary tree The index table contains 2^d positions, where d is the number of left-most bits currently being used to address the index table Initially, the table contains only one position, which points to a single bucket in use When this bucket fills, the table is doubled in size, a new bucket is created and the keys are rearranged

Ellis [14] has proposed a distributed extendible hashing technique As in a sequential system the hash structure consists of two parts: the directory component and the buckets It is this indirection provided by the directory that allows the buckets to be distributed to different sites in the distributed system and the directories to be replicated among the sites and managed by directory managers The buckets are linked to each other through a link field that allows recovery from restructuring operations The directory manager is essentially a server capable of handling multiple requests The bucket manager is a front end process that manages a disjoint set of buckets. An operation request on the hash table is sent to any directory manager which in turn forwards the request to a bucket manager after performing a directory lookup

The directory manager is then free to accept another request A bucket manager on receiving a request spawns a new slave process to service the request The directory manager has to propagate the update information to all the other directory managers therefore one problem is that its failure affects the entire system Fault tolerance capabilities are discussed that involve more messages in the system.

- **Two Phase Hashing**

A new hash algorithm for massively parallel systems is proposed by Yen and Bastani ([61]) In sequential systems, chaining gives the best performance, but in massively parallel processors, this leads to a high communication cost Linear probing, however, has a low communication cost

This algorithm called two-phase hashing, combines the chaining and linear probing concepts Here, a hash table with *in table chaining* is used the hash table keeps chains in the table itself instead of having other chain nodes If the number of elements hashed at each entry is known, then the final location of each element can be computed The first phase computes the number of elements that are to be hashed at each entry From this, the final location of each element is computed The next phase produces the real hashing where the data are forwarded to the hash entry From the hash entry the data are then forwarded to the starting location of the chain. The chain is thus searched A slight variation of the linear probing algorithm known as the hypercube hash algorithm is also discussed In this algorithm, the hash table is mapped directly to the processor space (i.e, the *i*th entry is assigned to processor *i*) Collisions are resolved by rehashing The difference between the above two algorithms is the method of computation of the rehashed location

- **Trie Hashing**

Trie hashing has been discussed by Litwin [40]. As in a normal hashing technique, the records are stored in buckets. The bucket addresses are computed with a dynamic trie of size proportional to the file size. The trie is a result of splits that cause buckets to overflow. The trie can be stored on the disk as subtrees for large files. Normally, because of the high branching factor, two levels are sufficient to store a one gigabyte file, therefore, two accesses are sufficient. The paper also proposes a method for the control of the bucket load factor of a trie hashing file. The distributed aspect of the design is not considered.

Linear Hashing for distributed files has been proposed by Litwin et al. [41]. It is useful for creating large files where the distribution of objects is necessary to exploit parallelism. It is suitable for creating scalable distributed data structures (SDDS). The mechanism is called LH* and an LH* file can grow to any size. A file is stored in a bucket at each server site. Since the bucket itself could be a single disk file, it is possible to create extremely large scalable files. Clients insert or retrieve objects from the file. Clients and servers are the nodes of a network and can be extended to any number of sites. A structure is termed SDDS if it can expand to new servers gracefully only when the currently used ones are efficiently loaded. There should be no master site that would make the system unreliable. Finally file access primitives should not be atomic actions. A simulation of the SDDS on a shared nothing multiprocessor showed that it takes one message (three in the worst case) per key insert and two messages (four in the worst case) for retrieval. They also showed that the average performance is close to optimal for both inserts and retrievals.

A family of order preserving scalable distributed data structures, namely RP* has been proposed by Litwin, et al [42]. To support range-queries and ordered traversals, conventional ordered data structures such as B trees are suitable. However, range-partitioning SDDSs provide for dynamic files on multicomputers. The fundamental algorithm builds the file with the same key space as a B tree but without the indices by using multicast. Two other algorithms enhance throughput of the network by adding the indices on either the clients, or on the clients and the servers simultaneously reducing the multicast.

Distributed file organization for disk resident files has been discussed by Viegrack et al. [58]. The focus of their work has been to achieve scalability (in terms of the number of servers) of the throughput and the file size while dynamically distributing data. Their results indicate that scalability is achieved at a controlled cost/performance.

Dictionary

A dictionary is a dynamic data structure that supports the following operations: insert, delete and search (lookup). It is one of the most fundamental data structures and is useful in many applications, such as natural language systems and database systems, for implementing symbol tables and pattern matching systems. In a conventional system an operation on the dictionary is a function of the number of elements. Pipelining the operations will give more parallelism, but this leads to a bottleneck for the more frequently accessed items. The bottleneck becomes severe as the number of processors increases. In a single processor environment, dictionaries are usually implemented as tree structures such as the AVL tree and the B^+ tree. The response of the sequential dictionary machines is a logarithmic function of the number of elements.

A sequential dictionary machine has been proposed that allows simultaneous and redundant accesses [15]. The objective of this system is to remove the sequential access bottleneck. The design consists of a sorting network and a binary tree with the data elements being stored at the leaf nodes. The accesses are sorted to form groups. The data elements are also ordered and made into groups so that the interaction with a group takes logarithmic time. The accesses within a group are sent to different groups of data elements. The binary tree serves to distribute the accesses.

An implementation of a distributed dictionary is described by Dietzfelbinger in [13]. The implementation is on a completely synchronized network of processors and is based on hashing. The keys to be inserted, deleted and searched are distributed to the processors via a hash function and processed using a dynamic hashing technique.

A distributed dictionary using B-link trees has been proposed [23]. This paper distributes the nodes of the tree among the processors. The interior nodes are replicated to improve parallelism and alleviate the bottleneck. The processor that owns a leaf owns all the nodes on the path from the root to the leaf. Restructuring decisions are made locally, thereby reducing the communication costs and increasing parallelism. The paper also deals with the problem of data balancing across the processors.

Another highly concurrent dictionary for parallel shared memory has been described by Parker [45]. This approach implements a dictionary independent of the underlying architecture. A new data structure called a sibling trie is used to implement the dictionary. Sibling tries, though based on trees, are a special kind of graph. The graph should be strongly connected so that every node is reachable from every other node. Multiple processes can search, insert, delete and update the data without creating hot spots. The advantage of using the sibling trie is that the search can start from any node, not necessarily the root, thereby reducing hot spots and providing alternate routes to a data item. The trie, a binary tree that implements a radix search, is the first component of the data structure and the second one is a sibling graph which connects nodes at the same level. Parker uses links to increase concurrency. The sibling graph is similar to the links used in a B-tree [49] and allows fast sequential access. The links in the sibling graph are used to traverse the entire structure, hence the diameter of the graph must be kept small. The paper presents an algorithm to perform search operations, but the distribution of the trie is not dealt with.

Peleg [46] has presented a detailed example of a distributed data structure. A compact dictionary structure, called BIN, is described. The BIN is based on a "flat" tree consisting of two levels: a central vertex serving as the directory and a collection of bins (each maintained at some vertex) that store the data in an orderly fashion. The paper also discusses the distribution of the central server and replication issues. Complexity issues and memory balancing are also addressed.

Search structures based on Linear Ordinary Leaves Structures (LOLS) family, (such as B^+ trees, K-D-B trees, etc.) have been proposed [43]. The paper addresses the problem of designing search structures to fit shared memory multiprocessor multidisk systems. The index of the structure is partitioned into a number of identical sub-indices (the sub-indices have the same structure and contents) which are stored in the shared memory while the data leaves that contain the data records are distributed across the processors. The design goal is to decrease the main memory consumption while having the same parallel processing capability, the same access time per operation and same disk utilization as other methods which use a single index structure.

1.3 Contributions of this Work

This dissertation addresses several issues such as fully distributing a B-tree, location independent naming of a node, data balancing and replication, among others.

Data partitioning also raises new issues such as allocating storage for the data, efficiency of access and balancing data among processors. The factors of concern for distributed storage are throughput, scalability and reliability. Most of these topics of interest are available in the current literature, but not in correlation with each other. Our work addresses all these issues.

We have developed a theoretical framework for replicating the interior nodes of the B-tree. Based on this, we have implemented two strategies of replication, namely full replication and path replication. The performance of these algorithms show that the path replication is better and is more scalable. We have developed several algorithms for data balancing a distributed replicated B-tree. We present the performance of our algorithms. An application of the work is the distributed extent tree, (dE-tree). We developed several data balancing algorithms for the distributed extent tree.

1.3.1 Structure of the Dissertation

We have organized this dissertation into two broad categories: theory and practical. In Chapter 2, we provide background on concurrent B-trees, the distributed B-tree and the distributed extent tree.

Chapter 3 provides the theoretical framework for replicating the nodes of a B-tree. In Chapter 4, we present the implementation design details. We present the underlying architecture and message passing mechanism for our implementation. We also present some generalized protocols that are common to all our data balancing algorithms. Finally, we discuss the portability of our implementation from the SUNs to the KSR, a shared memory parallel machine.

The performance of our replication and data balancing algorithms are presented in Chapter 5. Here, we discuss the replication strategies and discuss the results on their performance. We next discuss the various data balancing algorithms on the dB-tree and compare their performance. We also present the performance of the dE-tree.

We conclude the dissertation by summarizing the contribution of our work and providing some ideas about the direction for future research.

CHAPTER 2
SURVEY OF RELEVANT WORK

2.1 Introduction

In this chapter, we present some background on concurrent B trees, concurrent B-link algorithms, the distributed B-tree, and data balancing the distributed B-tree. We also provide a discussion of the paper by Johnson and Colbrook ([25]). They introduce a new balanced search tree algorithm for distributed memory systems. They use the B link tree as a basis for the distributed B-tree, the *dB tree*. To reduce the cost of maintenance of the distributed B tree, a *path replication* strategy is used, wherein if a processor owns a leaf node then it also owns all the nodes from the root to the leaf. The replication of the root at every processor enables operations to be initiated at any processor. The leaf level nodes are not replicated. The concept of *data balancing* has also been introduced to balance the load at all processors. They present some ideas on how data balancing can be implemented using distributed B link tree algorithms. Finally, they also show how the dB-tree algorithms can be used to build a data balanced distributed dictionary, the *dE-tree*.

2.2 Concurrent B-trees

Tree structures (in particular B trees) are suitable for creating indices. B trees of high order are desirable since they result in a reduction of the number of disk accesses needed to search an index. If the index has N entries, then a B tree of order $m = N + 1$ would have only one level. An insertion which causes a node to become too full splits the node and a restructuring of the tree is performed.

Current database designs necessitate the construction of databases which allow for concurrency of several processes. The original B tree algorithms were designed for sequential applications, where only one process accessed and manipulated the B tree. The main concern of these algorithms was minimising access latency. However, with the growth of processing power and the need for parallel computing, maximizing throughput has become important. The B tree is suitable for concurrent operations by allowing individual processes to perform independent operations.

Several approaches to concurrent access of the B tree have been proposed [7], [37], [44], [52]. All the algorithms share the problem of contention which can be categorized into two types: *data contention* and *resource contention*. Both lead to performance degradation.

- *Data contention*: All concurrent search tree algorithms require a concurrency control technique to keep two or more processes which access the B tree from interfering with one another. This contention is more pronounced at the higher levels of the tree. All algorithms proposed use some form of locking technique to ensure exclusive access to a node.

- *Resource contention*: Performance degradation is inevitable when several processes access a single resource in the system. In shared memory, this scenario occurs when more than one process contends for the same memory location. In a distributed architecture, contention occurs when one processor receives messages requesting access to a node from every other processor. Sagiv [49], and Lehman and Yao [37] use a link technique to reduce contention.

Parallel B-trees using multi version memory have been proposed by Wang and Weihl [60]. The algorithm is designed for software cache management and is suitable for cache-coherent shared memory multiprocessors. Every processor has a copy of the leaf node and the updates to the copies are made in a "lazy" manner. A multi version memory allows a process to read an "old version" of data. Therefore, individual read and write appear no longer atomic. A multi version memory thus allows a data read to progress concurrently with a data write. Also, "cache misses" are eliminated since no invalidation is done on writes and processes do not have to wait for update or do not invalidate messages from replicated copies.

Multi disk B trees have been proposed by Seeger and Larson [53]. They propose three different strategies for distributing the data stored on a B tree over multiple disks: record distribution, large page B-trees and page distribution. Local and global load balancing is also addressed. The main focus of the paper is the throughput of the system. Local load balancing is found to significantly reduce the response time for range queries.

2.2.1 Concurrent B tree Link Algorithm

A B-tree of order m is a tree that satisfies the following conditions.

1. Every node has no more than m children.

2. The root has at least two children and every other internal node has at least $m/2$ children.

3. The distance from the root to any leaf is the same.

A search for a key progresses recursively down from the root node. If the root node holds the key, the search stops, otherwise, it continues downward. An insert operation results in an insertion if the key is not already in the B-tree. If the node is full (i.e., an insertion would cause it to contain $m+1$ keys), the node splits and transfers half its keys ($m/2$) to the new sibling, and a pointer to the sibling is placed in the parent. If the insertion causes the parent to split, the split moves upward recursively. A delete searches for the key and removes it from the leaf node when found. If the node has less than $m/2$ keys, it is merged with either sibling. This technique is known as *merge-at-half* technique. A better option is to *free-at-empty* — delete nodes only when they are empty.

A variant of the B-tree known as the B^+-tree stores the data only at the leaf nodes. This structure is much easier to implement than a B-tree. A B-link tree is a B^+ tree in which every node has a pointer to its right sibling at the same level. The link provides a means of reaching a node when a split has occurred, thereby helping the node to recover from misnavigated operations. The B-link-tree algorithms have been found to have the highest performance of all concurrent B-tree algorithms [22]. In the concurrent B-link tree proposed by Sagiv [49], every node has a field that is the highest-valued key stored in the subtree.

A search operation starts at the *root* node and proceeds downwards. In this algorithm at most only one node is locked at any time. A search first places an **R** (read) lock on the root, then finds the correct child to follow. Next, the root node is unlocked and the child is **R** locked. Having reached a leaf node, the search finds the correct leaf node (i.e. the one whose highest value is greater than the key being searched for) by traversing the right links in a node. The search returns a success or failure depending on the presence of the key in the leaf node or not (2.1).

An insert operation works in two phases: a search phase and a restructuring phase [23]. The difference between the search phase of an insert operation and the search operation described above is that here the **R** lock on the leaf nodes is replaced by a **W** (exclusive write) lock. The key is inserted, if not already present, in the appropriate leaf. If the insert causes a leaf node to become too full, a split occurs and the restructuring begins as in the usual B-tree algorithm. Since the operations

```
struct tree-node root;
struct tree-node *findnextnode()
struct tree-node *findsibling()
boolean leaf
            success,  /* search is a success */
            failure   /* search is a failure */

Procedure search (v  n)
int v
struct tree-node *n,
{
            struct tree-node *node;

            node = root;
            Rlock (node)
            while (not leaf)
            {
                        child = findnextnode(node, v)
                        unlock (node)
                        node   child
                        Rlock(node)
            }

            /* traverse right links till correct leaf node is found */
            sib = findsibling (node, v),
            unlock (node)
            success = findkey (sib  v);
            if (success)
            {
                        *n = sib
                        return (success)
            }
            else
            {
                        *n = NULL;
                        return (failure)
            }
}
```

Figure 2 1 Search Algorithm for a B link Tree.

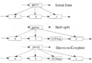


Figure 2.2 Half split Operation

hold at most only one lock at a time, restructuring must be separated into disjoint operations The first phase is to perform a *half-split* operation (Figure 2 2) During this phase, a new node the sibling is created and half the keys from the original node are transferred into it The sibling is put into the leaf list and the sibling pointers are adjusted appropriately The next phase is to inform the parent of the split Now the lock on the leaf node is released, the parent node is locked, and a pointer to the sibling is inserted into the parent During the time that the split occurs and the pointer is inserted into the parent, operations navigate to the sibling via the **link** and the **highest** fields in the node

On-the-fly node deletion is not supported in shared-memory multiprocessors. Several alternatives to on-the-fly deletion exist, including never deleting nodes, performing garbage collection or leaving the deleted nodes as stubs without deallocating them physically

2.3 The dB-tree

Johnson and Colbrook [23] present a distributed B tree suitable for message passing architectures. The interior nodes are replicated to improve parallelism and alleviate the bottleneck. The processor that owns a leaf owns all the nodes on the path from the root to the leaf. Restructuring decisions are made locally, thereby reducing the communication overhead and increasing parallelism. The paper also deals with the data balancing among processors.

The dB tree is built upon the concurrent B-link algorithms. In the dB-tree, the leaves are distributed among processors. The interior nodes are replicated among the processors. Every processor at a level has links to both its neighbours. Also, each node stores the distance from the leaves. Nodes of the dB-tree are given unique *tags*. A processor increments a node counter on the creation of a node. The tag is a concatenation of a node counter at a processor and the processor number. A translation table is used to access a node.

The operations *insert*, *delete* and *search* are defined on the dB tree. Corresponding to each operation, *actions* are performed on the nodes of the tree. A processor accepts messages from other processors for performing the operations. Misnavigated messages are routed to the correct processor. When a node becomes full, it "half splits". The double links of a node help in performing the half-split. Similarly when a node merges into another node or becomes empty, it must be deleted from the tree. A *half merge* procedure is used. All links to a merged node must be changed before a merged node can actually be deleted from the tree.

2.4 Replication

The multi version memory algorithm proposed by Wang and Weihl [80] reduce the amount of synchronization and communication needed to maintain replicated copies, thus reducing the effect of resource contention. Several algorithms have been proposed for replicating a node [8]. Lazy replication has been proposed by Ladin et al., for replicating servers [34]. The servers appear to be logically centralized in spite of their physical distribution. Replicas communicate information among themselves by lazily exchanging gossip messages. Johnson and Krishna [26] have proposed fixed copy and variable-copy algorithms for lazy updates on a distributed B tree.

2.4.1 Concurrency Control and Replica Coherency

All actions on a node are assumed to be performed atomically. The atomicity can be achieved by locking every copy of the node that is to be modified and blocking all reads and updates on the node. However, this is too restrictive. Johnson and Colbrook maintain replica coherency with far less synchronization and overhead. Only the modification is distributed to the copies, not the entire node contents. A node is never in a incorrect state, hence reading need not "block". Also, most modifying actions commute so the order in which they are performed does not matter. In chapter 3.2 we will see how two pending inserts at a parent can be performed in any order at the various copies of the parent.

However not all actions on a node can be performed in an arbitrary order. If an insert and a delete are pending on two copies of a full node, an insert being performed first leads to a split in the node in one copy while none in the other copy. The problem is the ordering of the split with the insert or delete. Johnson et al., present correctness criteria for the data structure.

They categorize actions on nodes as being *lazy*, *semi synchronous* or *synchronous* according to the amount of synchronisation required to perform the action. A lazy action does not need to synchronize with other lazy actions. A semi synchronous action must synchronize with some but not all other actions. A synchronous action is that which must be ordered with all other actions, or that requires communication with other nodes

Johnson and Krishna [26] present a framework for creating and analyzing lazy update algorithms The framework is used to develop algorithms that can manage a dB tree node The algorithm uses lazy insert actions and semi synchronous half split actions In addition the algorithm framework accounts for *ordered actions* to require that classes of actions are performed on a node in the order in which they are generated (i e the link-change actions are ordered)

2.5 Data Balancing

To avoid unbalanced storage space utilization at processors, it is necessary to perform data balancing on the processors The balancing also spreads the queries to the data structure evenly among processors It also provides equal memory and space utilization at each processor

Data balancing among processors has been studied by Johnson and Colbrook [23] They suggest a way of reducing communication cost for data balancing by storing neighboring leaves on the same processor When a processor decides that it has too many leaves it looks at a processor holding adjacent leaves If that processor accepts the leaves, the excess leaves are transferred If no neighboring processor is lightly loaded the heavily loaded processor looks for a lightly loaded processor and transfers the leaves

In the context of node mobility, object mobility has been proposed in Emerald [29] Objects keep forwarding information even after they have moved to another node and use a broadcast protocol if no forwarding information is available.

Lee et al. [36] have discussed a fault tolerant scheme for distributing queues. The scheme described by them provides dynamic fault tolerance, high availability and uniform load balancing with small storage space requirements and low communication. High availability is achieved by replication of the queue and each queue replica may be distributed over several sites. Consistency is maintained by two-phase locking. Small storage space is needed at each processor, since only segments of the queue may be kept at a processor. Since global broadcasting is not used, the communication overhead is low. However, every queue access requires communication to ensure global consistency. When a processor issues a queue operation, it sends a request to the processor containing the head or tail of the queue. On receiving the request, the current head or tail processor will lock up all other head or tail queue replicas, thereby ensuring consistency. If the processor which receives a request does not hold the head or tail, it forwards the request. The chasing continues until the processor holding the head or tail is found.

Ellis' algorithm [14] performs data balancing whenever a processor runs out of storage. Peleg [46] has studied the issue of data-balancing in distributed dictionaries from a complexity point of view, requiring that no processor store more than $O(M/N)$ keys, where M is the number of keys and N is the number of processors. In practice, this definition is simultaneously too strong and too weak because it ignores constants and node capacities.

In the dB-tree, the data balancing is performed by distributing the leaves among the processors. This requires communication among the processors each time a leaf moves to update sibling and parent links. Also, the number of interior nodes replicated is high. An alternative to the dB tree is the dE-tree.

2.6 dE-tree

To reduce the communication cost, Johnson and Colbrook suggest the dE-tree, also known as the distributed extent tree, where neighboring leaves are stored on the same processor. They define an *extent* to be a maximal length sequence of neighboring leaves that are owned by the same processor. When a processor decides that it owns too many leaves, it first looks at the processors who own neighboring extents. If the neighbor will accept the leaves, the processor transfers some of its leaves to the neighbor. If no neighboring processor is lightly loaded, the heavily loaded processor searches for a lightly loaded processor and creates a new extent.

Figure 2.3 shows a four processor dB-tree that is data balanced using the extents. The extents have the characteristics of a leaf in the dB-tree: they have an upper and lower range, are doubly linked, accept the dictionary operations, and are occasionally split or merged. The extent balanced dB-tree can be treated as a *dE-tree*. Each processor manages a number of extents. The keys stored in the extent are kept in some convenient data structure. Each extent is linked with its neighboring extent.

The extents are managed as the leaves in a dB tree. When a processor decides that it is too heavily loaded, it first looks at the neighboring extents to take some of its keys. If all neighboring processors are heavily loaded, a new extent is created for a lightly loaded processor. The creation and deletion of extents, and the shifting of keys between extents in the dE-tree correspond to splitting and merging leaves in the dB tree, and the index can be updated by using dB-tree algorithms.

Since a processor can store many keys, the index size is proportional to the number of processors. Also, index restructuring is greatly reduced as it takes place only after a large number of keys have been inserted or deleted.

The dE-tree can be used to maintain striped file systems ([27]).

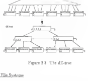


Figure 2 3 The dB-tree

2.8.1 Striped File Systems

Parallel file systems have been proposed to better match IO throughput to processing power A *parallel filesystem* is a file system in which the files are stored on multiple disks and the disk drives are located on different processors A common method for implementing a parallel filesystem is to use *disk striping* [51], in which consecutive blocks in a file are stored on different disk drives, each disk has its own controller A parallel striped file system, *Bridge* has been implemented on the BBN butterfly [33] A striped file can be appended (or prepended) to and maintain its structure However, a block can't be inserted into or deleted from the middle of the file, since doing so would destroy the regular striping structure of the file because of an out-of-order block or a gap A reorganization of the file is required Bridge however, does not support these operations In many applications, the most common operations on the file are "read" and "append", so striping reduces latency Certain other applications use "inserts" and "deletes" from the middle of the file

In the indexed striped file proposed by Johnson [27], the file consists of a single extent initially. An insert or delete calls for decisions to be made about re-organizing the extents. The inserts in the extent that cause a split correspond to the splitting of a node in our B tree and joining of two extents corresponds to a merging of the nodes. Thus, the dB-tree algorithms can provide an index structure which allows one to insert into or delete from a striped file, and further, as the striped extents are linked together, the file can be sequentially read in a highly parallel manner to provide fast random access. Direct access to the file is also fast.

The assumption is that the file is composed of *records*, each of which can be identified by a key which in turn can be ordered. This assumption is reasonable because the meaning of "insert this data after the 100 th block in the file" loses meaning when data blocks are being inserted and deleted concurrently.

The dE-tree is appropriate for a file index structure which allows insertions and deletions in the middle of a parallel striped file (if the records in the file are ordered), and that permits fast random access and highly parallel block reads. Instead of maintaining a single striped file, a sequence of independently striped extents is maintained, i.e. a striped file is broken into extents and an index into the extents is kept. The idea is that on an insertion or a deletion, either the extent can be reorganized or a new extent created. The dB tree index helps to manage the striped extents.

An example of an indexed striped file is shown in figure 2.4. The file is broken into a number of extents, each of which is independently striped across M disks (i.e., a *striped extent*). The extents are indexed by a dB-tree. The index is used for managing the extents, as well as for providing an index for random access.

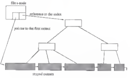


Figure 2 4 An Indexed Striped File

2.7 Conclusion

In this chapter, we have presented a background on concurrent B-trees and the distributed B-tree We have discussed the work done by Johnson and Colbrook ([23]) They present some ideas on the implementation of a distributed B tree and also present some techniques to avoid the root bottleneck by replication of the interior nodes Further, they discuss some ideas on data balancing the processors which hold the distributed B-tree. Concurrency control and replica coherency are also addressed To reduce the cost of communication for data-balancing, they suggest the distributed extent tree The dE-tree manages extents instead of individual keys Johnson ([27] provides a discussion of how the dE-tree can be used for a practical application of striped file systems In the next chapter, we provide a theoretical framework of the algorithms for replication of the distributed B-tree

CHAPTER 3
REPLICATION ALGORITHMS

3.1 Introduction

When addressing large volumes of data, there is a danger of memory bottlenecks, where all processors access the same data item stored at one processor. For example, one of the problems with a distributed search structure is that since all accesses to the data have to pass via the root node, the root node becomes a bottleneck and overwhelms the node which stores it (as noted in [6]). It also creates excessive message traffic in the network towards the processor which holds the root node of the search structure. This is known as resource contention and can be solved by replication. Allowing multiple copies of often accessed nodes distributes the work load among the components of the system. Replication while providing redundancy, availability and improving concurrency, however, introduces consistency problems previously not present. A method of achieving consistency is to guarantee that all operations take place in the same order at all the sites of the distributed system.

Several algorithms have been proposed for replicating a node [8]. These, however do so at the cost of concurrency since they require synchronisation and thus create significant communication overhead. Lazy replication has been proposed by Ladin and Laskov for replicating servers [34]. The servers appear to be logically centralized, in spite of their physical distribution. Replicas communicate information among themselves by lazily exchanging gossip messages. This, however, creates the following problem. Consider two different operations, a and b, that are causally related but executing at different replicas A and B. If operation b is dependent on the previous one, a, the replica which receives b, i.e., B, does not have enough information about it to proceed. The replica, B has to delay the operation of b until b receives all the updates it depends on.

Techniques exist to reduce the cost of maintaining replicated data and for increasing concurrency. Ladin, Liskov, and Shira propose *lazy replication* for maintaining replicated servers [34]. Lazy replication uses the dependencies that exist in the operations to determine if a server's data is sufficiently up-to-date to execute a new request. Several authors have explored the construction of non-blocking and wait-free concurrent data structures in a shared-memory environment [17]. These algorithms enhance concurrency because a slow operation never blocks a fast operation.

In this chapter, we present an approach to maintaining distributed data structures which uses *lazy updates*, which take advantage of the semantics of the search structure operations to allow for scalable and low overhead replication. Lazy updates can be used to design distributed search structures that support very high levels of concurrency. The alternatives to lazy update algorithms (vigorous updates) use synchronization to ensure consistency.

Lazy update algorithms are similar to lazy replication algorithms because both use the semantics of an operation to reduce the cost of maintaining replicated copies. The effects of an operation can be lazily sent to the other servers, perhaps on piggybacked messages. The lazy replication algorithm blocks an operation until the local data is sufficiently up-to-date. In contrast, a non-blocking wait-free concurrent data structure never blocks an operation. The lazy update algorithms are similar in that the execution of a remote operation never blocks a local operation; hence, they are a distributed analogue of non-blocking algorithms.

Lazy updates have a number of pragmatic advantages over more vigorous replication algorithms. They significantly reduce maintenance overhead. They are highly concurrent, since they permit concurrent reads, reads concurrent with updates, and concurrent updates (at different nodes). Since lazy updates avoid the use of synchronization, they are much easier to implement than vigorous update algorithms.

Despite the benefits of the lazy update approach, implementors might be reluctant to use it without correctness guarantees. We develop a correctness theory for lazy updates so that our algorithms can be applied to other distributed search structures. We demonstrate the application of lazy updates to the dB-tree, which is a distributed B^+ tree which replicates its interior nodes for highly parallel access[23].

We present three algorithms, the last of which can implement a dB-tree which never merges nodes and performs data balancing on leaf nodes (we have previously found that never merging nodes results in little loss in space utilization [21], and data balancing on the leaf level is low overhead and effective [30]). The methods we present can be applied to other distributed search structures, such as hash tables [14].

Before we terminate this introduction, we should mention some useful characteristics of lazy updates. First, when a lazy update is performed at one copy of a node, it must also be performed at the other copies. Since the lazy update commutes with other updates, there is no pressing need to inform the other copies of the update immediately. Instead, the lazy update can be piggybacked onto messages used for other purposes, greatly reducing the cost of replication management (this is similar to the *lazy replication* techniques [34]). Second, index node searches and updates commute, so that one copy of a node may be read while another copy is being updated. Further, two updates to the copies of a node may proceed at the same time. As a result, the dB-tree not only supports concurrent read actions on different copies of its nodes, it supports concurrent reads and updates, as well as concurrent updates.

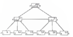


Figure 3 1 A dB tree

3.2 Replication

All operations start by accessing the root of the search structure If there is only one copy of the root, then access to the index is serialized. Therefore, we want to replicate the root widely in order to improve parallelism. As we increase the degree of replication however the cost of maintaining coherent copies of a node increases Since the root is rarely updated, maintaining coherence at the root isn't a problem A leaf is rarely accessed but a significant portion of the accesses are updates As a result wide replication of leaf nodes is prohibitively expensive

In the dB tree the leaf nodes are stored on a single processor We apply the rule that if a processor stores a leaf node, it stores every node on the path from the root to that leaf An example of a dB-tree which uses this replication policy is shown in Figure 3 1 The dB-tree replication policy stores the root everywhere, the leaves at a single processor, and the intermediate nodes at a moderate level of replication As a result an operation can be initiated at every processor simultaneously, but the effects of updates are localized As a side effect an operation can perform much of its searching locally, reducing the number of messages passed

The replication strategy for a dB-tree helps to reduce the cost of maintaining a distributed search structure, but the replication strategy alone is not enough If every node update required the execution of an available-copies algorithm [8], the overhead

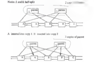


Figure 3.2 Lazy inserts

of maintaining replicated copies would be prohibitive. Instead, we take advantage of the semantics of the actions on the search structure nodes and use *lazy updates* to maintain the replicated copies inexpensively.

We note that many of the actions on a dB-tree node commute. For example, consider the sequence of actions which occurs in Figure 3.2. Suppose that nodes A and B split at "about the same time." Pointers to the new siblings must be inserted into the parent, of which there are two copies. A pointer to A' is inserted into the first copy of the parent and a pointer to B' is inserted into the second copy of the parent. At this point, the search structure is inconsistent, since not only does the parent not contain a pointer to one of its children, but the two copies of the parent don't contain the same value.

The tree in Figure 3.2 is still usable, since no node has been made unavailable. Further, the copies of the parents will eventually converge to the same value. Therefore, there is no need for one insert action on a node to synchronize with another insert action on a node. The tree is always navigable, so the execution of an insert doesn't block a search action. We call node actions with such loose synchronization requirements *lazy updates*.

3.3 Correctness of Distributed Search Structures

Shasha and Goodman [56] provide a framework for proving the correctness of non-replicated concurrent data structures. We make extensive use of their framework in order to discuss operation correctness. We delete most details here to save space, but we note that if the distributed analogue of a link type search structure algorithm follows the Shasha Goodman link algorithm guidelines, it will produce strict serializable (or linearizable) executions. However, we would like the distributed search structure to satisfy additional correctness constraints. For example, when a distributed computation terminates, every copy of a node should have the same value. Performing concurrency control on the copies is discussed in the following sections.

3.4 Copy Correctness

We intuitively want the replicated nodes of the distributed search structure to contain the same value eventually. We can ensure the coherence of the copies by serializing the actions on the nodes (perhaps via an "available-copies" algorithm [8]). However, we want to be *lazy* about the maintenance. In this section, we describe a model of distributed search structure computation and establish correctness criteria for lazy updates.

A node of the logical search structure might be stored at several different processors. We say that the physically stored replicas of the logical node are *copies* of the logical node. We denote by $copies_t(n)$ the set of copies that correspond to node n at (global snapshot) time t.

An operation is performed by executing a sequence of actions on the copies of the nodes of the search structure. Thus, the specification of an action on a copy has two components: a final value c' and a subsequent action set SA. An action that modifies a node (an *update* action) is performed on one of the copies first, then is relayed to the remaining copies. We distinguish between the *initial* action and the *relayed* actions. Thus, the specification of an action is

$$a^t(p, c) = (c', SA)$$

When action a with parameter p is performed on copy c, copy c is replaced by c' and the subsequent actions in SA are scheduled for execution. Each subsequent action in SA is of the form (a_i, p_i, c_i), indicating that action a_i with parameter p_i should be performed on copy c_i. If copy c_i is stored locally, the processor puts the action in the set of executable actions. If c_i is stored remotely, then the action is sent to the processor which stores c_i. If the action is a *return value* action, a message containing the return value is sent to the processor that initiated the operation. If the final value of $a(p, c)$ is c for every valid p and c, then a is a *non-update* action, otherwise, a is an *update* action. The superscript t is either i or r, indicating an initial or a relayed action. We also distinguish initial actions by writing them in capitals, and relayed actions by writing them in lowercase (i.e., I and i for an insert).

In order to discuss the commutativity of actions, we will need to specify whether the order of two actions can be exchanged. If action a^t with parameter p can be performed on c to produce subsequent action set SA, then the action is *valid*, otherwise the action is *invalid*. We note that the validity of an action does not depend on the final value.

An algorithm might require that some actions must be performed on all copies of a node, or on all copies of several nodes "simultaneously." Thus, we group some action sequences into *atomic action sequences*, or AAS. The execution of an AAS at a copy is initiated by an *AAS_start* action and terminated by an *AAS_finish* action. A copy may run one or more AAS simultaneously. An AAS will commute with some actions (possibly other AAS_start actions) and conflict with others. We assume that the node manager at each processor is aware of the AAS action conflict relationships, and will block actions that conflict with currently executing AAS. The AAS is the distributed analogue of the shared memory lock, and can be used to implement a similar kind of synchronization. However, lazy updates are preferable.

3.4.1 Histories

In order to capture the conditions under which actions on a copy commute, we model the value of a copy by its history (as in [18]). Formally, the *total history* of copy $c \in copies_l(n)$ consists of the pair (I_c, A'_c), where I_c is the initial value of c and A'_c is a totally ordered set of actions of c. We define correctness in terms of the update actions, since non update actions should not be required to execute at every copy. The *(update) history* of a copy is a pair (I_c, A_c) where I_c is the same initial value as in the total history, and A_c is A'_c with the non update actions deleted (and the order on the update actions preserved). To remove the distinction between initial and relayed actions, we define the *uniform* history, $U(H)$ to be the update history H with each action a^t replaced by a. Finally, we will write the history of copy c (I_c, A_c) as $H_c = I_c \prod_{i=1}^{m} a_c^i$, where $A_c = \{a_c^1, a_c^2, \ldots, a_c^m\}$.

Suppose that $H_c = I_c \prod_{i=1}^{m} a_i$ and that I_c is the final value of $H' = I' \prod_{j=1}^{k} a'_j$. Then $H_c^* = (I' \prod_{j=1}^{k} a'_j) \prod_{i=1}^{m} a_i$ is the *backwards extension* of H_c by H'. It is easy to see that H_c and H_c^* have the same value, and the last m actions in H_c^* have the same subsequent action sets as the m actions in H_c. When a node is created, it has an initial value, I_n. When a copy of a node is created it is given an initial value, which we call the *original value* of the copy. This initial value should be chosen in some meaningful way, and will typically be equivalent to the history of the creating copy, or to a synthesis of the histories of the existing copies. In either case, the new copy will have a backwards extension which corresponds to the history of update actions performed on the copy. If a copy of a node is deleted, then we no longer need to worry about the node contents. We denote a set of all initial update actions performed on node n by m_n.

We recall that an action on a copy is valid if the action on the current value of the copy has its associated subsequent action. A history is *valid* if action a_i is valid on $I_c \prod_{j=1}^{i-1} a_j^c$ for every $i = 1, \ldots, m$. The *final value* of a history is the final value of the last action in the history. Two histories are *compatible* if they are valid, have the same final values, and have the same uniform updates. If H_1 and H_2 are compatible, then we write $H_1 \equiv H_2$.

Our correctness criteria for the replica maintenance algorithms are the following:

Compatible History Requirement. A node n with initial value I_n and update action set M_n has *compatible histories* if, at the end of the computation C,

1. every copy $c \in copies(n)$ with history H_c has a backwards extension B_c such that the update actions in $H_c' = B_c | H_c$ contains exactly the actions in M_n.
2. every backwards extension H_c' can be rearranged to form H_c^* such that $U(H_c^*) = U(H_{c'}^*)$ for every $c, c' \in copies(n)$, and every H_c^* is valid.

If an algorithm guarantees that every node has a compatible history, then it meets the *compatible history requirement*.

Complete History Requirement: If every subsequent action issued appears in some node's update action set, then the computation meets the *complete history requirement*. If every computation that an algorithm produces satisfies the complete history requirement, then the algorithm satisfies the *complete history requirement*.

Ordered History Requirement. We define an *ordered action* as one that belongs to a class τ such that all actions of class τ are time-ordered with each other (we assume a total order exists). A history H is an *ordered history* if for any ordered actions $h_1, h_2 \in H$ of class τ, if $h_1 <_\tau h_2$ then $h_1 < h_2$ in H. An algorithm meets the *ordered history requirement* if every node has a compatible history that is an ordered history.

The compatible history requirement guarantees that every node is single-copy equivalent when the computation terminates. We note that our condition for rearranging uniform histories is a condition of the subsequent action sets rather than a condition of the intermediate values of the nodes. The copies need only to have the same value at the end of the computation, but the subsequent actions can't be posthumously issued or withdrawn without a special protocol.

The complete history requirement tells us that we must route every issued action to a copy. A deleted node is conceptually retained in the search structure to satisfy the complete history requirement. The ordered history requirement lets us remove explicit synchronization constraints on the equivalent parallel algorithm by shifting the constraints to the copy coherence algorithm.

3.4.2 Lazy Updates

An update action must be performed on all copies of a node. With no further information about the action, it must be performed via an AAS to ensure that the conflicting actions are ordered in the same way with all copies. However, some actions commute with other almost all other actions, removing the need for an AAS. In Figure 3.2, the final value of the node is the same at either copy, and the search structure is always in a good state. Therefore, there is no need to agree on the order of execution. We provide a rough taxonomy of the degree of synchronization different updates require.

Lazy Update We say that a search structure update is a *lazy update* if it commutes with all other lazy updates, so synchronization is not required.

Semi-synchronous update Other updates are almost lazy updates, but they conflict with some other actions. For example, the actions may belong to a class of ordered actions. We call these *semi-synchronous* updates. A semi synchronous action requires special treatment, but does not require the activation of an AAS.

Synchronous Update A synchronous update requires an AAS for correct execution. We note that the AAS might block only a subclass of other actions, or might extend to the copies of several different nodes.

3.5 Algorithms

In this section, we describe algorithms for the lazy maintenance of several different dB tree algorithms. We work from a simple fixed-copies distributed B-tree to a more complex variable-copies B tree, and develop the tools and techniques we need along the way. For all of the algorithms we develop, we assume that only search and insert operations are performed on the dB tree. In addition, we assume the network is reliable, delivering every message exactly once in order.

3.5.1 Fixed Position Copies

For this algorithm, we assume every node has a fixed set of copies. This assumption lets us concentrate on specifying lazy updates. Every node contains pointers to its children, its parent, and its siblings. When a node is created, its set of copies are also created, and copies of the node are never destroyed.

A search operation issues a search action for the root. The search action is a straightforward translation of the action that a shared memory B link tree algorithm takes at a node. An insert operation searches for the correct leaf using search actions, then performs an insert action on the leaf. If the leaf becomes too full, the operation restructures the dB tree by issuing half-split and insert actions The insert action adds a new key at the leaves and adds a pointer to a child in the non-leaf nodes. The half split action creates a new sibling (and the sibling's copies), transfers keys from the half split node to the sibling, modifies the node to point to the sibling, and sends an insert action to the parent

The first step in designing a distributed algorithm is to specify the commutativity relationships between actions.

1 Any two insert actions on a copy commute As in Sagiv's algorithm [49], we need to take care to perform out-of-order inserts properly

2 Half split operations do not commute Since a half-split action modifies the right sibling pointer the final value of a copy depends on the order in which the half splits are processed

3 Relayed half split actions commute with relayed inserts, but not with performed initial inserts Suppose that in history H_p, initial insert action $I(A)$ is performed before a half split action s that removes A's range from p Then, if the order of I and s are switched I becomes an invalid action A relayed insert action has no subsequent actions and the final value of the node is the same in either ordering Therefore, relayed half splits and relayed inserts commute

4 Initial half split actions don't commute with relayed insert actions One of the subsequent actions of an initial half split action is to create the new sibling The key which is inserted either will or won't appear in the sibling depending on whether it occurs before or after the half split

By our classification methods, an insert is a lazy update and a half-split is a semisynchronous update If the ordering between half-splits and inserts isn't maintained the result is lost updates (see Figure 3 3) We next present two algorithms to manage fixed copy nodes To order the half-splits both algorithms use a *primary copy* (PC), which executes all initial half-split actions (non-PC copies never execute initial half split actions, only relayed half splits) The algorithms differ in how the insert and half split actions are ordered The synchronous algorithm uses the order of half splits and inserts at the primary copy as the standard to which all other copies must adhere The semi synchronous algorithm requires that the ordering at the primary copy be consistent with the ordering at all other nodes (see Figure 3 4)

We do not require that all initial insert actions are performed at the PC, so copies might find that they exceed their maximum capacity However, since each copy is maintained serially, it is a simple matter to add overflow blocks

I1
I2
I3
I4
s1
C1
I2
I1
i3
S1
i4
PC
I3
i2
I
i4
s1
C2

Problem.

If S1 transfers reduces the range of the node to exclude I4 s key then I4 s key is lost.
The PC ignores an out-of-range relayed insert.
The copies discard I4 s key when they perform the relayed split.

Figure 3 3 An example of the lost-insert problem

Synchronous Splits

Algorithm. An operation is executed by submitting an action, and each action generates subsequent actions until the operation is completed An operation is executed by executing its B-link tree actions as discussed previously Thus, all we need to do is specify the execution of an action at a copy The synchronous split algorithm uses no AAS to ensure that splits and inserts are ordered the same way at the PC and at the non-PC copies (see Figure 3 4)

Half-split Only the PC executes initial half-split actions Non-PC copies execute relayed half-split actions When the PC detects that it must half split the node, it does the following

1. Performs a split_start AAS locally This AAS blocks all initial insert actions, but not relayed insert or search actions

2. The PC sends a split_start AAS to all of the other copies.

3. The PC waits for acknowledgments from all of the copies of the AAS

4. When the PC receives all of the acknowledgments, it performs the half split, creating all copies of the new sibling and sending them the sibling's original value

5. The PC sends a split_end AAS to all copies and performs a split_end AAS on itself

When a non-PC copy receives a split_start AAS, it blocks the execution of initial inserts and sends an acknowledgment to the PC The executions of further initial insert actions on the copy are blocked until the PC sends a split_end AAS When the copy processes the split_end AAS, it modifies the range of the copy, and the right sibling pointer, discards pointers no longer in the node's range, and unblocks the initial insert actions

Insert When a copy receives an initial insert action it does the following

1. Checks to see if the insert is in the copy's range If not the insert action is sent to the right sibling

2 If the insert is in range, and the copy is performing a split AAS the insert is blocked, otherwise

3 The insert is performed and relayed insert actions are sent to all of the other copies

When a copy receives a relayed insert action, it checks to see if the insert is in the copy's range If so, the copy performs the insert. Otherwise, the action is discarded

Search When a copy receives a search action, it examines the node's current state and issues the appropriate subsequent action

We note that since non-PC copies can t initiate a half split action, they may be required to perform an insert on a too-full node Actions on a copy are performed on a single processor, so it is not a problem to attach a temporary overflow bucket The PC will soon detect the overflow condition and issue a half split, correcting the problem

Theorem 1 The synchronous split algorithm satisfies the complete, compatible and ordered history requirements

Proof We observe that the fourth link algorithm guideline is satisfied, so that whenever an action arrives at a copy, its parameter is within the copy's inreach Therefore, the synchronous split algorithm satisfies the complete history requirement

Since there are no ordered actions, the synchronous split algorithm vacuously satisfies the ordered history requirement

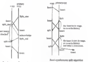
We show that the synchronous algorithm produces compatible histories by showing that the histories at each node are compatible with the uniform history at the primary copy First, consider the ordering of the half split actions (a half split is performed at a node when the split_end AAS is executed) All initial half split actions are performed at the PC, then are relayed to the other copies Since we assume that messages are received in the order sent, all half splits are processed in the same order at all nodes.

Consider an initial insert I and a relayed half-split s performed at non PC copy c If $I < s$ in H_c then I must have been performed at c before the AAS_start for s arrived at c (because the AAS_start blocks initial inserts) Therefore, I's relayed insert i must have been sent to the PC before the acknowledgment of s was sent By message ordering i is received at the PC before S is performed at the PC, so $i < S$ in H_{PC} If $s < I$ in H_c, then $S < i$ in H_{PC}, because $S < s$ and $I < i$ (due to message passing causality) □

We note that this algorithm makes good use of lazy updates For example, only the PC needs an acknowledgment of the split AAS If every channel of communication between copies had to be flushed, a split action would require $O(|\text{copies}(n)|^2)$ messages instead of the $O(|\text{copies}(n)|)$ messages this algorithm uses. Furthermore, search actions are never blocked

Semi synchronous Splits

We can greatly improve on the asynchronous split algorithm For example, the synchronous split algorithm blocks initial inserts when a split is being performed Furthermore $3 * |\text{copies}(n)|$ messages are required to perform the split By applying the "trick" of *rewriting history*, we can obtain a simpler algorithm which never blocks insert actions and requires only $|\text{copies}(n)|$ messages per split (and therefore is optimal)



Synchronous split algorithm blocks new inserts while a split executes.

Semi-synchronous split algorithm never blocks inserts, instead rewrites history to ensure compatible histories.

Figure 3 4 Synchronous and semi synchronous split ordering

The synchronous-split algorithm ensures that an initial insert I and a relayed split s at a non PC node are performed in the same order as the corresponding relayed insert i and initial split S are performed at the PC, with the PC ordering setting the standard We can turn this requirement around and let the non PC copies determine the ordering on initial inserts and initial splits, and place the burden on the PC to comply with the ordering.

Suppose that the PC performs initial split S_i then receives a relayed insert i_c from c where I_c was performed before s at c (see Figure 3 4). We can keep H_{PC} compatible with H_c by rewriting H_{PC}, inserting i_c before S in H_{PC} If i_c's key is in the PC's range, then H_{PC} can be rewritten by performing i_c on the PC Otherwise, i_c's key should have been sent to the sibling that s created Fortunately the PC can correct its mistake by creating a new initial insert with i_c's key, and sending it to the sibling This is the basis for the semi synchronous split algorithm

Algorithm. The semi-synchronous split algorithm is the same as the synchronous split algorithm, with the following exceptions:

1. When the PC detects that a split needs to occur, it performs the initial split (creates the copies of the new sibling, etc.), then sends relayed split actions to the other copies.

2. When a non-PC copy receives a relayed split action, it performs the relayed split.

3. If the PC receives a relayed insert and the insert is not in the range of the PC, the PC creates an initial insert action and sends it to the right neighbor.

Theorem 2 The semi-synchronous split algorithm satisfies the complete, consistent and ordered history requirements.

Proof. The semi-synchronous algorithm can be shown to produce complete and ordered histories in the same manner as in the proof of Theorem 1.

We need to show that all copies of a node have compatible histories. Since relayed inserts and relayed splits commute, we need only consider the cases when at least one of the actions is an initial action. Suppose that copy c performs initial insert I after relayed split s. Then, by message causality, the PC has already performed S, so the PC will perform i after S.

Suppose that c performs I before s and PC performs i after S. If i is in the range of PC after S, then i can be moved before S in H_{PC} without modifying any other actions. If i is no longer in the range of PC after S, then moving i before S in H_{PC} requires that S's subsequent action be modified to include sending i to the new sibling. This is exactly the action the algorithm takes. □

Theorem 2 shows that we can take advantage of the semantics of the insert and split actions to lazily manage replicated copies of the interior nodes of the B-tree. In the next section, we observe a different type of lazy copy management which also simplifies implementation and improves performance.

3.5.2 Single-copy Mobile Nodes

In this section, we briefly examine the problem of lazy node mobility. We assume that there is only a single copy of each node, but that the nodes of the B-tree can migrate from processor to processor (typically, to perform load balancing). When a node migrates, the host processor can broadcast its new location to every other processor that manages the node (as is done in Emerald [29]). However, this algorithm requires large amounts of wasted effort and doesn't solve the garbage collection problems.

The algorithms we propose inform the node's immediate neighbors of the new address. In order to find the neighbors, a node contains links to both its left and right sibling, as well as to its parent and its children. When a node migrates to a different processor, it leaves behind a *forwarding address*. If a message arrives for a node that has migrated, the message is routed by the forwarding address. We are left with the problem of garbage-collecting the forwarding addresses (when is it safe to reclaim the space used by a forwarding address?). As with the fixed copies scenario, we propose an eager and a lazy algorithm to satisfy the protocol. We have implemented the lazy protocol, and found it effectively supports data balancing [30].

The eager algorithm ensures that a forwarding address exists until the processor is guaranteed that no message will arrive for it. Unfortunately, obtaining such a guarantee is complex and requires much message passing and synchronization. We omit the details of the eager algorithm to save space.

Suppose that a node migrates and doesn't leave behind a forwarding address If a message arrives for the migrated node then the message clearly has misnavigated This situation is similar to the misnavigated operations in the concurrent B link protocol, which suggests that we can use a similar mechanism to recover from the error We need to find a pointer to follow If the processor stores a tree node, then that node contains the first link on the path to the correct destination So the error-recovery mechanism is to find a node that is 'close' to the destination and follow that set of links

The other issue to address is the ordering of the actions on the nodes (since there is only one copy, every node history is vacuously compatible) The possible actions are the following insert split, migrate, and link change. The link change actions are a new development in that they are issued from an external source, and need to be performed in the order issued

Algorithm. Every node has two additional identifiers, a version number and a *level* The version number allows us to lazily produce ordered histories The level, which indicates the distance to a leaf, aids in recovery from misnavigation An operation is executed by executing its B link tree actions, so we only need to specify the execution of the actions

Out of range. When a message arrives at a node, the processor first checks if the node is in range This check includes testing to see if the node level and the message destination level match If the message is out of range or on the wrong level, the node routes it in the appropriate direction

Migration When a node migrates,

1 all actions on the node are blocked until the migration terminates

2. A duplicate copy of the node is made on a remote processor, (with the exception that the version number increases by 1)

3. a link change action is sent to all known neighbors of the node.

4. the original node is deleted

Insert. Inserts are performed locally

Half-split Half splits are performed locally by placing the sibling on the same processor and assigning the sibling a version number one greater than the half split node's. An insert action is sent to the parent, and a link change action is sent to the right neighbor

Link change When a node receives a link-change action, it updates the indicated link only if the update's version number is greater than the link's current version number. If the update is performed, the new version number is recorded

Missing Node If a message arrives for a node at a processor, but the processor doesn't store the node, the processor performs the out-of range action at a locally stored node. If the processor doesn't store a search structure node, the action is sent to the root

Theorem 8 The lazy algorithm satisfies the complete, compatible and ordered history requirements

Proof There is only a single copy of a node, so the histories are vacuously compatible. Each action takes a good state to a good state so every action eventually finds its destination. Therefore the algorithm produces complete histories.

The only ordered actions are the link change actions. The node at the end of a link can only change due to a split or a migration. In both cases, the node's version number is incremented. When a link change action arrives at the correct destination, it is performed only if the version number of the new node is larger than the version number of the current node. If the update is not performed, the node's history is rewritten to insert the link change into its proper place. Let l be a link change action that is not performed, and let l be an ordered action of class $\mathcal{L}$. Let a_i be the ordered action of class $\mathcal{L}$ in H_c that is ordered immediately after l (there is no a_j such that $l <_{\mathcal{L}} a_j <_{\mathcal{L}} a_i$). We rewrite H_c to be $H'_c = I_c \left(\prod_{k=1}^{i-1} a_k\right) \ l \ \left(\prod_{k=i}^{|H_c|} a_k\right)$. Thus, the history can be rewritten so that it remains valid. □

We note that an implementation of the lazy single-copy algorithm can use forwarding addresses to improve efficiency and reduce overhead. The forwarding addresses are not required for correctness, so they can be garbage-collected at convenient intervals.

3.5.3 Variable Copies

In this scenario, we assume that leaf level nodes can migrate, and that processors can join and leave the replication of the index nodes (so we can use this algorithm to implement a never merge dB tree). We assume that the leaf nodes are not replicated, and that the PC of a node never changes.

The lazy algorithm that we propose combines elements of the lazy fixed copy and migrating node algorithms by using lazy splits, version numbers, and message recovery.

To allow for data-balancing, we let the leaf level nodes migrate. The leaf level nodes aren't replicated, so we can manage them with the lazy algorithm for migrating nodes (section 3.5.2). We want to maintain the dB tree property that if a processor owns a leaf node, it has a copy of every node on the path from the root to the leaf. If a node obtains a new leaf node, it must join the set of copies for every node from the root to the leaf which it does not already help maintain. If the processor sends off the last child of a node, it unjoins the set of processors which maintain the parent (applied recursively). When a processor joins or unjoins a node replication, the neighboring nodes are informed of the new cooperating processor with a *link-change* action. To facilitate link change actions, we require that a node have pointers to both its left and right sibling. Therefore, a split action generates a link change subsequent action for the right sibling, as well as an insert action for the parent.

We assume that every node has a PC that never changes (we can relax this assumption). The primary copy is responsible for performing all initial split actions for registering all join and unjoin actions. The join and unjoin actions are analogous to the migrate actions. Hence, every join or unjoin registration increments the version number of the node. The version number permits the correct execution of ordered actions, and also helps ensure that copies which join a replication obtain a complete history (see Figure 3.5). When a processor unjoins a replication, it will ignore all relayed actions on that node and perform error recovery on all initial action requests.

Algorithm.

Out-of range If a copy receives an initial action that is out-of range, the copy sends the action across the appropriate link. Relayed actions that are out of range are discarded.

Insert 1. When a copy receives an initial insert action, it performs the insert and sends relayed insert actions to the other node copies that it is aware of. The copy attaches its version number to the update.

2. When a non PC copy receives a relayed insert, it performs the insert if it is in range, and discards it otherwise.

3. When the PC receives a relayed insert action, it tests to see if the relayed insert action is in range.

   (a) If the insert is in range, the PC performs the insert. The PC then relays the insert action to all copies that joined the replication at a later version than the version attached to the relayed update.

   (b) If the insert is not in range, the PC sends an initial insert action to the appropriate neighbor.

Split 1. When the PC detects that its copy is too full, it performs a half split action by creating a new sibling on several processors, designating one of them to be the PC, and transferring half of its keys to the copies of the new sibling. The PC sets the starting version number of the new sibling to be its own version number plus one. Finally, the PC sends an insert action to the parent, a link change action to the PC of its old right sibling, and relayed-split actions to the other copies.

2. When a non PC copy receives a relayed half-split action, it performs the half split locally.

Join When a processor joins a replication of a copy, it sends a join action to the PC of the node. The PC increments the version number of the node and sends a copy to the requester. The PC then informs every processor in the replication of the new member and performs a link change action on all of its neighbors.

Unjoin When a processor unjoins a replication of a node, it sends an unjoin action to the PC and deletes its copy. The processor discards relayed actions on the node and performs error recovery on the initial actions. When the PC receives the unjoin action it removes the processor from the list of copies, relays the unjoin to the other copies, and performs a link change action on all of its neighbors

Relayed join/unjoin When a non PC copy receives a join or an unjoin action, it updates its list of participants and its version number

Link-change A link change action is executed using the migrating-node algorithm

Missing-node When a processor receives an initial action for a node it doesn't manage, it submits the action to a 'close' node, or returns the action to the sender

Theorem 4 The variable copies algorithm satisfies the complete, compatible, and ordered history requirements

Proof We can show that the variable-copies algorithm produces complete and ordered histories by using the proof of Theorem 3 If we can show that for every node n, the history of every copy $c \in copies(n)$ has a backwards extension H_c' whose uniform update actions are exactly M_n, then the proof of theorem 2 shows that the variable copies algorithm produces compatible histories

For a node n with primary copy PC let A_i be the set of update actions performed on PC when the PC has version number i When copy c is created, the PC updates its version number to j and gives c an initial value $I_c = I_n B_j$, where B_j is the backwards extension of I_c to I_n and contains all uniform update actions in A_1 through A_{j-1} The PC next informs all other copies of the new replication member After a copy c' is informed of c, c' will send all of its updates to c. The copy c' might perform some initial updates concurrent with c's joining $copies(n)$ These concurrent updates are detected by the PC by the version number algorithm and are relayed to c Therefore, at the end of a computation, every copy $c \in copies(n)$ has every update in M_n in its uniform history Thus, the variable copies algorithm produces compatible histories □

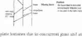


Figure 3.5 Incomplete histories due to concurrent joins and inserts

3.6 Conclusion

In this chapter, we have discussed the following

- Replication Algorithms
- Lazy Updates on a dB tree
- Correctness theory for Lazy Updates

We present algorithms for implementing lazy updates on a dB-tree, a distributed B tree The algorithms can be used to implement a dB tree which never merges empty nodes and performs data balancing on the leaves (we have previously found that the free-at-empty policy provides good space utilization [21] and that leaf level data balancing is effective and low overhead [30]) We provide a correctness theory for lazy updates, so lazy update techniques can be used to implement lazy updates on other distributed and replicated search structures [14] Lazy updates, like lazy replication, permit the efficient maintenance of the replicated index nodes Since little synchronization is required lazy updates permit concurrent search and modification of a node, and even concurrent modification of a node. Finally, distributed search structures which use lazy updates are easier to implement than more restrictive algorithms because lazy updates avoid the use of synchronization The next chapter presents the details of our implementation of the distributed B tree

CHAPTER 4
IMPLEMENTATION

4.1 Introduction

A distributed environment consists of processors capable of communicating with each other through messages. We implemented the distributed B-tree on a general network architecture, a LAN network comprised of SPARC workstations. Every processor is capable of communicating with other processors and has sufficient amount of local storage. Each processor acts as a server responding to messages from other processors.

4.2 Design Overview

The B tree is distributed by partitioning the nodes of the tree across a network of processors. The network of processors communicate by sockets (a Unix Internetwork message passing scheme). To provide a user interface, we integrated Xwindows in our design. In this design, there is an overall B tree manager, called the anchor, which overlooks all the B tree operations. The anchor is responsible for creating new processes on different processors when necessary. Every processor is individually responsible for the nodes it maintains.

On each processor we have a *queue manager* and a *node manager*. The queue manager receives messages from remote processors and maintains them in a queue. The node manager takes messages from the queue and performs the operations (specified in the message) on the various nodes at that processor. This distinction of process

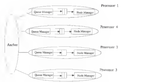


Figure 4.1 The Communication Channels

functionality into the queue manager and the node manager enables the node manager to be independent of the inter-processor communication method. The queue manager and the node manager at a processor communicate via the inter process communication schemes supported by UNIX, namely message queues (Figure 4.1).

4.2.1 Anchor Process

The anchor is responsible for initializing the B tree. In addition, the anchor receives update messages from external applications and sends them to the appropriate processor. Each processor is responsible for the decision it makes concerning the tree structure it holds. In the current implementation, the anchor makes the decision if two or more processors are involved. In order to do so, the anchor must have a picture of the global state of the system. The B-tree processing will continue while the anchor makes its decision, so the global picture will usually be somewhat out of date. Our algorithms take this fact into account.

The anchor begins building the tree by selecting a processor (the *root processor*) to hold the root of the tree. The node manager at the root processor has a socket connection to the anchor. Update operations are passed to the root processor and percolate down to the leaf level, where the decisive action of the operation is performed. The dashed lines in figure 4.1 represent temporary communication channels established between two processors for the transfer of nodes, which will be described in a later section.

4.2.2 Node Structure

Logically adjacent nodes may not reside at the same processor; hence, a parent/child/sibling pointer may refer to a node at some other processor. Also, nodes cannot be uniquely identified by the local address. Every node in the B tree has a name associated with it that is not dependent on the location of the node. This mechanism for naming nodes is known as location-independent naming of nodes. A typical node would have a parent pointer, the children pointers and the sibling pointers. In addition to having the highest value in itself, the node must also keep the high value of it's logical neighbors. This will enable a node to determine if the operation is meant for itself or destined for either of its siblings.

- **Location Independent Naming of a Node**

  Whenever a node is created, it is given a name that is unique among all processors. For instance, the node name may be a combination of the processor number that creates it and the node identifier within the processor. A hashing mechanism is used to translate between node names and physical node addresses. When a node bub moves from processor *A* to processor *B*, it retains its name. The advantage of this mode of naming nodes is that a parent child or sibling node that references the node **bob** need not know the exact address of **bob** in processor *B*.

A further advantage of location independent naming is when the nodes are replicated. All copies of a node on different processors have the same name. So, the primary and secondary copies of a node can keep track of each other easily.

4.2.3 Updates

Our implementation is primarily concerned with the update operations: *inserts* and *deletes*.

- **Inserts:**

  An insert operation at a leaf processor inserts the key in the appropriate node, say *n*. If the insertion of a key causes the node, *n* to become too full, the node splits by creating a new sibling, *s* and moves half the keys from the original node, *n* to the new sibling *s*. The parent node *p* of node *n* is informed of this split by sending a message to the processor the parent it resides on. The message contains the name of the new sibling *s* and the modified high and low values of *n* and *s*. To improve parallelism and reduce the number of messages in the system, the child processor does not wait for an acknowledgement from the parent processor, instead, we use the B link tree protocol.

  When the parent node, *p* receives a split message from the child, it adds the node *s* as a new child, and adjusts the high and low values of its children *n* and *s*. If the addition of the child *s* causes the parent node *p* to become full, it splits into *p* and *np*. The keys transfer takes place as at the lower level and a split message from the parent *p* travels to its parent *gp* and the process may recurse upward till the root The children of p are not informed immediately of the split in the parent, so some of the children of p (those transferred to node np) will have pointers to p instead of to np. These obsolete parent pointers are updated when a message arrives from the parent np to its child If the child s has the parent pointer as p and receives a message from np, it uses the source node information (in this case np) in the message to update its parent pointer to np Our design can tolerate the "lazy" update of these pointers since a message from the child s to the old parent p, will find the correct new parent, np by using the sibling pointers at the parent node p

If an insert causes the parent to split, the message percolates up towards the root node In the event that the root node splits, a new root has to be created The processor holding the root node creates a new node and makes that the new root However it informs all other processors that the tree height has increased

- **Deletes**

The delete operations pose more complications, as deletion of keys means shifting the responsibility of a key range between two nodes A delete operation removes the key from a leaf level node, e g , node u. The restructuring actions the algorithm takes depends on whether we have implemented a *free-at empty* or *merge-at half* B-tree Merging across processors involves too much overhead in terms of synchronization and messages and thus is not cost efficient So, if the neighbors are on the same processor, then the merge at half protocol is used otherwise, the node is allowed to become empty (i e , free-at empty protocol is used)

The problem that occurs when nodes can split as well as merge is that some actions can be performed twice at some copies, leading to inconsistency. This transpires when an action occurs at the PC before the split and at a non PC copy after the merge.

When the key ranges of interior nodes change due to merging, then care must be taken to synchronize the inserts or deletes with the splits and merges. Let us consider this scenario. Suppose there are three copies of a node, $c1$, $c2$, and PC (Figure 4.2). Let the initial insert of key k, $I(k)$ be performed at c1. The relayed insert $i(k)$ is relayed to $c2$ and the PC. Before the relayed insert $i(k)$ reaches the PC, the node n has split into n' and s. The relayed insert $i(k)$ at the PC is forwarded to the sibling s as $I'(k)$ and is performed there. This is now relayed as $i'(k)$ to the copies $c1$ and $c2$. The copy $c2$ performs $i'(k)$ on s. Now suppose $D(k)$ is performed on s at PC. Subsequently, relayed deletes $d(k)$ are performed on s at copies $c1$ and $c2$. Let the nodes n' and s merge now to form n'' and s (where n'' contains the range of k). Now, the relayed insert $i(k)$ (from copy $c1$) arrives at $c2$ and k is inserted in n'', losing the action $d(k)$. The copies now of n'' at $c1$ and the PC do not have the key k, but that at $c2$ contains the key k. Thus the key k is inserted twice and never deleted from $c2$. (If the action $i(k)$ had arrived at $c2$ before the merge, then the node n' for which it was intended would not contain the range and hence would be discarded, leaving all copies consistent.)

i. **Free-at-empty**

A node n that becomes empty does not get deleted until its neighbors update their links. A processor that receives a sibling empty message blocks deletes and sends an acknowledgement after it has set the link.

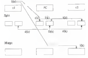


Figure 4 2 Duplicate actions due to merges

After the acknowledgements are received from both neighbors the space is freed. The node pointer must also be deleted from the parent A message is sent to the parent node and n is marked as deleted However, the node remains in the doubly linked list with its siblings until an acknowledgement arrives from the parent This ensures that no further updates to the node n will be received, so n is removed from the list and its space is reclaimed In the interval before the acknowledgement is received, any operations to the deleted node n are sent to its siblings (as appropriate) If a node is asked to delete a pointer that does not exist, (as the relayed insert has not yet arrived at that copy) but is in its key range, the delete action is delayed until the corresponding insert action arrives Thus, a node has to remember *delayed deletes*

2 **Merge-at-half**

In addition to deleting nodes that are empty, we have incorporated a merge protocol to implement *merge at half* If the removal of a key reduces n to less than half its maximum capacity, the node shares its keys either to the right or to the left The idea here is to keep the nodes equally full If the right or left neighbor has more than half the keys, the excess is shared with the node n

> The transfer of keys between two adjacent leaves, must be recorded at the parent The parent is made aware of the key range in its child subtree so that future updates would be directed properly

When the parent node receives a message to delete a child node, it removes the pointer to the child On receiving a change in the key range message from a child, the parent changes the highest value of the child A change in the parent may cause one of the above situations so the algorithm is applied recursively (4 3) If a delete message reaches the root processor, it checks to see if it has only two children If so, one of them is deleted and it is left with one child. A message is sent to the anchor to shrink the tree The anchor makes the only child of the root the new root of the shorter tree It also removes the old root node and deallocates the processor holding the old root node.

To obtain a better understanding of how these protocols work, let us look at the algorithms in figures 4 3 through 4 6 When a processor receives a delete message, the message travels to the appropriate leaf node and then the procedure 4 3 is invoked In this algorithm, the key v is deleted from the node n that resides on processor p The contents of node n have changed, so the state of node n is decided by invoking the algorithm **decide_state** (figure 4 4) Procedure **decide_state** may return any of the following values (4 4)

- INITVAL In this case the root node has been reached and so the delete process is completed A *relayed_delete* message is sent to all copies of the node

- EMPTY_LOCAL The parent of node n resides on the same processor, p as node n so the parent is updated of the key deletion in node n, and the process continues recursively upwards

- EMPTY_REMOTE The parent of node n does not reside on the same processor, so a message is sent to the processor holding the parent node, indicating that the node n has become empty and to remove the child pointer to n.

- MERGE_RIGHT The right neighbor of node n resides on the same processor, p, so the nodes n and its right neighbor share the keys among themselves 4.5

- MERGE_LEFT The left neighbor of the node n resides on the same processor, p, so the nodes n and its left neighbor share the keys among themselves 4.6

- NO_MERGE. If the node n is neither empty nor less than half-full, then a merge cannot be done So the siblings of the node are updated and the parent of the node n is updated of the new high and low values of the node n, If the parent resides on the same processor Otherwise, a message is sent to the parent on some other processor to update node n's values A *relayed_delete* message is sent to all copies of the node

```
Procedure Recursive_delete(n, v) {
done = FALSE
while (!done) {
    done = TRUE;
    pos = position of key v in the node n
    remove_key(n, v)
    state = decide_state(n)
    switch (state) {
        case INITVAL
            send_relay_delete (n, v)
            break
        case EMPTY_LOCAL
            localparentupdate_empty(n, v)
            n = n->parent;
            done = FALSE
            break
        case EMPTY_REMOTE
            send_toparent_empty(n)
            break
        case MERGE_RIGHT
            perform_merge_right(n)
            break
        case MERGE_LEFT
            perform_merge_left(n)
            break
        case NO_MERGE
            update_siblings(n)
            if (n->parent.proc != CUR_PROC)
                send_toparentnewhigh(n, gethighest(n), gethighest(n))
            else
                localparentupdate(n,gethighest(n), gethighest(n)),
            send_relay_delete(n),
            break
        default
            break
        } }
```

Figure 4 3 Recursive_Delete Algorithm

```
Procedure decide_state(n)
struct node *n  {
struct node *next  *prev
int extra
if (n->parent_proc == INITVAL)
    return (INITVAL)
if (empty_node(n))
    if (n->parent_proc -= CUR_PROC)
        return (EMPTY_LEGAL)
    else
        return (EMPTY_ABNGTR);
else {
    if (n->right_proc == INITVAL && n->left_proc == INITVAL)
        return (NO_MERGE)
    if (half_node(n)) {
        if ((n->right_proc != CUR_PROC && n->right_proc != INITVAL) &&
            (n->left_proc != CUR_PROC && n->right_proc != INITVAL))
                return (NO_MERGE)
        else {
            next = n->link
            if (next != NULL) {
                if (nodelength(next) + nodelength (n) <= 2*MAXCHILD -1)
                    extra = nodelength(next)
                else
                    if (nodelength (next) > MAXCHILD)
                    extra - (nodelength(next) - MAXCHILD)/2
                    if (extra > 0)
                        return (MERGE_RIGHT)  }
            else {
                prev = n >leftlink
                if (prev != NULL) {
                    if (nodelength (prev) + nodelength(n) <= 2*MAXCHILD-1)
                    extra = nodelength(prev)
                    else
                    if (nodelength (prev) > MAXCHILD)
                        extra = (nodelength(prev) - MAXCHILD)/2;
                        if (extra > 0)
                            return (MERGE_LEFT), }
            } }
    }}
return (NO_MERGE),
}
```

Figure 4.4 Procedure Decide_State for Deletes

```
Procedure perform_merge_right (n)
struct node *n  {
oldhigh = gethighest(n);
next = n->link
nexthigh = gethighest(next)
if (next->proc == n->proc)
    empty = merge_right(n);
if (empty) {
    p = next->parent;
    if (p->proc == CUR_PROC)
        recursive_delete(p, nexthigh)
    else
        send_toparent_empty(next);}
newhigh = gethighest(n);
if (n->level > LEAF)
    update_parent_ofchildren(n);
update_siblings(n, next);
send_relay_merge(n, next);
if (n->parent_proc != CUR_PROC)
    send_toparentnewhigh(n, oldhigh, newhigh)
else
    localparentupdate(n, oldhigh, newhigh)
if (!empty) {
    if (next->parent_proc != CUR_PROC)
        send_toparent_newhigh(next, gethighest(next),gethighest(next)),
    else
        localparent_updatenewhigh(next, gethighest(next),gethighest(next)) }
}
```

Figure 4.5 Procedure Perform_merge_right for Deletes

```
Procedure perform_left_merge (n)
struct node *n  {
prev = n->leftlink
prevhigh = gethighest(prev)
empty = merge_left(n  prev)
if (empty) {
    if (prev->parent_proc == CUR_PROC)
        recursive_delete(prev->parent, prevhigh)
    else
        send_toparent_empty(prev)  }
else {
    if (prev->parent_proc == CUR_PROC)
        localparent_updatenewhigh(prev)
    else
        send toparent_newhigh(prev)  }
if (n->level > LEAF)
    update_parent_ofchildren(n)
update_siblings(n, prev)
send_relay_merge(n, prev)
if (n >parent_proc != CUR_PROC)
    send_toparentnewhigh(n  gethighest(n), gethighest(n))
else
    localparentupdate(n,gethighest(n), gethighest(n))
}
```

Figure 4 6 Procedure Perform_merge_left for Deletes

4.3 Data Balancing the B-tree

We have addressed the need for distributing the B-tree. Distributing the tree arbitrarily implies that some processors may have many nodes (due to splits). Hence, when there are plenty of nodes in the system, the processors run the risk of losing storage capacity. Hence, for efficient use of storage and other resources, it is necessary to balance the load among processors. We will be discussing the various algorithms for data balancing in the next chapter. However, certain inherent issues, such as methods for dealing with out-of-order messages caused by delays introduced by the underlying network, and low overhead synchronization of tree restructuring, will be discussed here. Methods for node mobility essential for data balancing and resource sharing are also discussed. We have developed algorithms for dynamic data-load balancing that use the mechanisms of node mobility. In this chapter, we present some issues pertaining to this balancing. Other issues that arise from load balancing are mechanisms for node mobility and out of order information handling.

The fundamental issue in load balancing is the actual process of moving a node between processors. This is termed the *node migration* mechanism and is common to all of our algorithms for load balancing.

Another important concern is the out-of-date information that the processors have. Since processors do not have up-to-date information about every other processor in the system, they must rely on the old information to make decisions. When an overloaded processor wishes to unload some of its nodes to another processor, it selects a receiving processor (how this is done will be explained in the next chapter) and follows a *negotiation protocol* to determine the exact the number of nodes to transfer. This will be discussed in greater detail in section 4.4.

The node migration algorithm should address the following questions:

1. **Who is involved?** Should the sender and all other processors be locked up until all pointers to the node in transit get updated? If this is so, parallelism would be lost. How should we achieve maximum parallelism?

2. **When is everyone informed?** Once a node has been selected for migration, how and when is every other processor informed of its new address? In the interim in which node movement takes place and the other related processors are informed, what happens to the updates that come for the node in transit? How do they get forwarded?

3. **How is Obsolete information handled?** When a node moves it sends an update link message to related processors. Suppose the update message for link change gets delayed and the node moves for a second time. The second update message may reach a reach a processor before the first one, what approach should one take to resolve this problem.

Our algorithm addresses these problems and provides solutions to them.

In the context of node mobility, object mobility has been proposed in Emerald [29]. Emerald is an object-based language which places emphasis on the mobility of objects. Objects in Emerald can be data objects or process objects and the distribution is adaptive to dynamically changing loads. Here, every object has a forwarding address comprised of a timestamp and address. Every time the object moves, the address and the timestamp is updated. If an object moves from node A to node B, only node A and node B are updated. When node C addresses the object at node A, the message is forwarded to node B. Finally, node B responds to the message and sends the message back to C with its new address piggybacked. Objects keep forwarding information even after they have moved to another node and use a broadcast protocol if no forwarding information is available.

4.3.1 Node Migration Algorithm

For the following discussion on the node migration mechanism, let us assume the node manager at a processor wishing to download its nodes has been notified of a recipient processor that is willing to accept nodes. The actual method by which this is done will be explained in a later section.

After the node manager is informed of a recipient for its excess nodes, it must decide which nodes to send. This may be based on various criteria of distribution. After selecting a node, the node manager begins the transfer. This procedure is explained by providing solutions to the problems posed in the introduction.

Who is involved? Our solution to the first problem is aimed at maintaining parallelism to the maximum extent possible by involving only the sender and the receiver during node movement. We have designed an *atomic handshake* and *negotiation protocol* for the node migration. Since nodes are uniquely named and a node retains its name when moved between processors, there is no need for acknowledgments in our algorithm. After the node selection is done, the sending processor (henceforth called the *sender*) establishes a communication channel with the *receiver* and a negotiation protocol follows. In the negotiation protocol, the sender and the receiver come to an agreement as to how many nodes are to be transferred. After a decision has been reached the sender sends a node, updates the forwarding information in the node and transfers the next node. A node that has been sent is tagged as in transit and no operations are performed on that node at the sender (Alg. 2).

When is everyone informed? The sender and receiver update all locally stored pointers to the transferred nodes. If the related nodes are on different processors (other than the sender and receiver), the receiver sends link update messages to them. If in the meantime some messages arrive for this node at the sender (since all processors are not yet aware of the migration) the messages are forwarded to the new address At specific intervals of time the nodes marked in transit are deleted and their storage reclaimed

Since we do not require acknowledgments for the link changes, it is possible that a message will arrive for the deleted node In this case, the node manager forwards the message to a local node that is "close" to the intended address The message then follows the B-link tree search protocol to reach its destination. In our current implementation, we are guaranteed that the processor stores either a parent of the deleted node, or another node on the same level as the deleted node The significance of this deleted node recovery protocol is that we can lazily inform neighboring nodes of a moved node's new address This protocol is rarely invoked, since most messages for the transferred node are handled by the forwarding address.

How is Obsolete information handled? The final problem is how to deal with out-of-order messages arriving at a processor In any network, one cannot guarantee the messages are delivered in the same order as sent The inherent delays in the underlying network cause messages to be sent out of order and sometimes even be lost Messages from a single source to a single destination arrive in the same order sent However, there is no order imposed on messages from multiple sources to a destination The question then is how does the system tolerate delayed and even lost messages The above problem can be translated to our B tree as shown by the following example (Figure 4 7)

Suppose node a moves from processor A to Processor B Consider node p, which resides on processor P and contains a link to node a When node a moves to processor B, an update message is sent to node p at processor P Before this message reaches P, processor B decides to move node a to processor C and C sends an update message to P Suppose the message from C reaches P before the message from B If node p

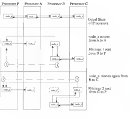


Figure 4 7 Node Migration

at P updates the node address to that of C and then to B, then node *p* at P has the wrong address for node *s*

In our design, we have a *version number* for every node of the tree. A node has a version number 1 when it gets created for the first time (unless it is the result of a split). Every time a node moves its version number is incremented, and when a node splits, the sibling gets a version number one greater than that of the original node. Every pointer has a version number attached and each link update message contains the version of the sending node. When node *r* receives a link update message from *s*, *r* will update the link only if *s*'s version number is equal to or greater than the link version number. In the above example, the version number of node *s* on processor A is initially 1. On moving to processor B, the version number changes to 2. The update message to P from B contains the version number 2. The next update message sent to P from C has the version number 3. Now since this last message reaches P first, node *p* at processor P notes that its version number for node *s* is 1. Since 3 >= 1, node *p* updates node *s*'s address, version number and processor number. Now, the message from B that contains version number 2 arrives. But now node *p* has version number 3 for node *s*, hence the version numbers do not match and the message is ignored. So delayed messages that arrive out of order at a processor are ignored.

Our numbering handles out-of-order link changes due to split actions also. Reliable communications guarantee that messages generated at a processor for the same destination arrive in the order generated, and when nodes move to different processors the version number of the nodes is incremented, so messages regarding link changes are processed in the order generated

4.4 Negotiation Protocol

In all our algorithms, no extra messages are sent to inform the other processors of a change in the current status of a processor. Thus, if the number of nodes at a processor increases or decreases due to splits or merges, other processors are not aware of it. Neither is the anchor process informed about these changes, the reason being to avoid excess network traffic. However, not informing others leads to stale information, where the anchor and the processors have old and outdated information about other processors. Now, in the load balancing algorithm when the anchor has to decide with whom an overloaded processor must share its data, it finds another processor based on the outdated information. We will show in the next chapter that our load balancing algorithms perform very well, in spite of old information because of the *negotiation protocol*.

We have designed an *atomic handshake protocol* for the negotiation. During the interim that the sending processor decides to share some of its data and a receiver processor is chosen, either by the anchor (in centralized loadbalancing) or by itself (in distributed load balancing with probing), the status of both processors may have changed. So, after the receiving processor is selected, the *sender* and the *receiver* enter into negotiation wherein they update the status of each other and decide exactly how many nodes to share. The negotiation involves only these two processors and hence other processors are not hindered. Once negotiation is completed, node transfer takes place. It should be noted that no messages are sent to other processors informing them of the negotiation or change in the status of the sender and the receiver.

4.5 Portability

Finally, we have ported our implementation to the KSR, a shared memory multiprocessor machine with 96 processors that supports message passing by providing

BSD sockets ([31]) The porting of our implementation shows that our systems is portable and easily scalable to a large number of processors

4.6 Conclusion

To conclude, this chapter has addressed the following

- Design issues for implementation
- Data balancing the dB tree and the fundamental protocols necessary
- Portability of the implementation

In this chapter we have discussed the implementation of the distributed B tree on a network of Sparc stations and the processes needed to manage the dB tree Update operations, insert, search and delete are performed on the B tree We have presented how these operations are performed and what complications the delete operations present and how we overcome them To facilitate data balancing on the distributed B-tree, we have introduced the convention of naming nodes so that a node retains its name between processors We will see that this node naming also is useful when replicating nodes at various processors, since all copies of a node have the same name

We have presented two mechanisms fundamental to data balancing, namely, the node-migration algorithm for the actual movement of nodes between processors and the negotiation protocol to overcome the effect of outdated information Methods by which our algorithms and protocols tolerate out-of-order messages introduced because of network delays are also presented

Finally, to study the portability and scalability of our implementation, we ported it to the KSR, a large scale shared memory multiprocessor system In the next chapter we discuss the algorithms for replication and load balancing and present performance results

CHAPTER 5
PERFORMANCE

5.1 Introduction

In this chapter we present the various algorithms for replication and data balancing and discuss their performance in detail. Experiments using the two strategies for replication, namely *full replication* and *path replication* were conducted. Results show that path replication will create a scalable distributed B-tree. We validated the tree scalability by simulating a large scale distributed B tree and performing large-scale experiments on it. Several load balancing algorithms have been developed and their performance measured. The observations reflect that all our load-balancing algorithms incur very little overhead while achieving a good data balance. We also discuss the performance of several load balancing algorithms on the dE-tree. Three algorithms, namely random, merge and aggressive merge algorithms, have been developed for data balancing on the dE-tree, and of these we find that aggressive algorithm makes the dE-tree scalable. Timing measurements have been conducted on our implementation of the dB tree to study the response times and throughput of our system. We present these results in this chapter. Using the data from the simulation experiments, we present an analytical performance model of the dB-tree and the dE-tree. We find that both algorithms are scalable to large numbers of processors.

5.2 Replication

In this section, we describe two algorithms for maintaining consistency among the copies of nodes. Based on the theoretical framework presented in Chapter 3, we have incorporated two replication strategies in our implementation. Our implementation of the Fixed Position copies algorithm is termed **Full Replication** and that of Variable copies is **Path Replication**. We will briefly discuss the algorithms and the implementational issues in sections 5.2.1 and 5.2.2.

When the nodes of the B-tree are replicated, an obvious concern is the consistency and coherency of the various replicated copies of a node. Subsection 5.2.3 will present the mechanism by which our implementation maintains coherent replicas.

5.2.1 Full Replication Algorithm

The Fixed position copies algorithm ([26]) assumes every node has a fixed set of copies. An insert operation searches for a leaf node and performs the insert action. If the leaf becomes full, a half split takes place. In this algorithm, the Primary Copy, PC performs all initial half-splits and sends a *relayed split* to the other copies. Any *initial inserts* at a non PC copy are kept in overflow buckets and adjusted after the *relayed split*.

In our implementation, the B-tree is distributed by having the leaf level nodes at different processors. Leaf level nodes are not replicated and only these nodes are allowed to migrate between processors. Whenever a leaf node migrates to a new processor (one that currently stores no leaves), the index levels of the tree are replicated at that processor. Consistency among the replicated nodes is maintained by the primary copy of a node sending changes to all its copies.

Once the entire tree has been replicated, only consistency changes need to be propagated to this new processor.

- **Algorithm** The decision to replicate the tree is made after a processor (sender) downloads some of its leaf level nodes to another processor (receiver). After the leaves are transferred, the sender checks to see if the receiver has received leaf nodes for the first time If so, the receiver obviously does not have the index levels, so the tree has to be replicated at the receiver The sender then transfers the tree (index levels) it currently holds Henceforth, only consistency maintenance messages are necessary to maintain the tree at this processor

5 2.2 Path Replication Algorithm

In the Variable-copies algorithm ([26]), different nodes have different number of copies. A processor that holds a leaf node also holds a path from the root to that leaf node. Hence, index level nodes are replicated to different extents A processor that acquires a new leaf node may also get new copies of index level nodes and such a processor then joins the set of node copies for the index level nodes Similarly, a processor will 'unjoin' a node when it has no copies of the node's children

In our path replication algorithm whenever a leaf node migrates to a different processor, entire path from the root to that leaf is replicated at this processor However, if the processor holds a leaf and a new sibling migrates to that processor, only the parent nodes not already resident at this processor are replicated All link changes are again handled by the primary copy of a node. When a new copy of a node is created, the processor sends a 'join' message to all the copies of the node In the interim of the node copy being created at the processor and the 'join' message reaching a processor, any messages about this node copy are forwarded by the primary copy of the node to this new copy A processor that sends away all the leaf nodes of a parent will no longer be eligible to hold the path from the root to that leaf node In this case the processor has to do an 'unjoin' for all its nodes on the path from the root to the leaf

- **Algorithm** Our algorithm for path replication is asynchronous, based on a handshaking protocol. When two processors have interacted in the load balancing protocol, a decision has to be made concerning the path from the root to the migrated leaves. Either the sending or receiving processor can request that the path be sent to the receiver. In our algorithm, the receiver determines what ancestor nodes are needed after receiving new leaves. It then sends requests to the processors holding the primary copies of the ancestor to get the paths. As the receiving processor takes the responsibility of obtaining the path, the sending processor is free to continue. The receiving processor cannot do much anyway until it receives the path, so no time is wasted. Once the path is obtained, the receiving processor can handle operations (inserts and searches) on its own.

5.2.3 Replica Coherency

The operations the current implementation handles are searches and inserts. A **search** operation is the same as an **insert** operation, except a key is not inserted. A search returns a success or failure and does not cause any further relayed messages to be issued. An operation on the distributed B-tree can be initiated on any processor. Since the index levels are fully or partially replicated at all processors, a change in a node copy at any processor must be informed to all processors that hold a copy of that node. Every processor that stores a copy of a node must be aware of all the inserts on that node. An insert operation to a node could result in a split, so all processors must be informed about the split. This is done in the following way.

- **Insert** An **insert** operation can be performed on any copy of a node. After performing the insert, the processor sends a **relayed insert** to all other processors that hold a copy of the node. When a processor receives a relayed insert, it performs the insert operation locally.

- Split A **split** operation is first performed at a leaf. If the local parent exists on the same processor, the split is informed at the local parent. If the split at any level results in a split at the parent level, then a **relayed split** is sent to all processors that hold a copy of the parent node. Otherwise, a **relayed insert** is sent.

5.2.4 Performance

Here, we compare the performance of full replication and path replication strategies for replicating the index nodes of a B-tree.

Experiments, Results and Discussion **Experiment Description.** In the experiment, 15 000 keys were inserted; statistics were gathered at 5000 key intervals. The B tree is distributed over 4 to 12 processors. Each node in the B-tree has a maximum fanout of 8, and average fanout of 5. We observed the number of times a path request has been made by a processor, the number of times that a load balancing request had to be reissued (to avoid deadlock), with priority being given to the path request. We collected statistics as to how many consistency messages are needed to maintain the distributed, replicated B tree, how widely the index nodes are replicated on each processor, and finally how many nodes each processor stores at the end of the run.

The *Message Overhead* (Figure 5 1) graph shows the number of messages needed to maintain the replicated B-tree. We see that in case of full replication, the number of messages for a 4 processor B tree is around 9000 and for 12 processors it is around

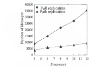


Figure 5 1. Full versus Path Replication Message Overhead

35000 (i.e the message overhead has increased linearly as the number of processors) However, for a path replicated B tree, for 4 processors around 3800 messages are needed and for 12 processors only 9300 messages are needed not even a linear increase.

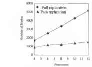


Figure 5 2 Full versus Path Replication Space Overhead

The *Space Overhead* (figure 5 2) graph shows the number of nodes stored at all processors at the end of a run The graph is similar in nature to the message overhead graph In this graph we consider only the index nodes that account for the excess storage at each processor (the leaf nodes remaining nearly the same for all processors) as the number of processors increase For full replication, we see that for a 4 processor B tree the number of index nodes stored is 1700, whereas for a 12 processor B tree the number of nodes is 5200, a nearly three-fold increase In case of a path replicated B-tree, the number of index nodes stored over the entire tree for 4 processors is 900 and for 12 processors is 1550, not even a two-fold increase

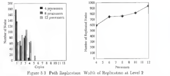


Figure 5 3 Path Replication Width of Replication at Level 2

The *Width of replication at level 2* (Figure 5 3) graphs show how widely level 2 index nodes are replicated at each processor for a path replicated B tree We selected level 2 since activity takes place at the leaf level, 1, and affects mostly at level 2. The bar chart shows the number of nodes in the B tree that have i copies, where i varies from 1 to 5, with the concentration being nodes with 1 copy at 4 processors The other chart, number of replicated nodes versus processors, shows that even as we increase the number of processors, the level 2 index nodes are not widely replicated at all processors, with there being 597 copies for a 4 processor system and only 944 copies for 12 processors.

Path replication causes low restructuring overhead, but can require a search to visit many processors for its execution We measured the number of hops required for the search phase of the insert operation after 5000 inserts were requested in an 8 processor distributed B-tree. Full replication required an average of 88 messages per search, and path replication required 1 29 messages per search (additional overhead of 41 messages)

From the above observations, we see that a path replicated distributed B tree performs better than a fully replicated one and is highly scalable (Figure 5 3)

5.3 Data Balancing

We have performed data balancing on the dB tree and the dE-tree We will discuss the algorithms and the performance of the two separately

5.3.1 The dB tree

The results obtained from the implementation of a replicated B tree led us to explore other algorithms for data balancing on a replicated B tree. The experiments with the replication algorithms led us to conclude that a path replicated B tree was more scalable than a fully replicated B tree. Hence we simulated a path replicated distributed B tree Our objective is to develop data balancing algorithms and also to observe their performance and overhead incurred

Algorithms

In the current design, a limit is placed on the maximum number of nodes of the tree that a processor can hold termed as the **threshold.** In addition each node has a **soft limit** (75 * threshold) on the number of nodes This represents a warning level indicating a need for distribution of the nodes Whenever a node splits, the current number of nodes is checked against the soft limit. If the current number of nodes exceeds the soft limit the processor must distribute some of the nodes it has to some other processor Our algorithms are characterized by the method by which the receiver processor is selected

- Centralized Data Balancing

  One approach is a *semi-centralized* one, where the anchor is responsible for choosing the receiver The overloaded processor requests the anchor for another processor that can share its excess load

  The anchor has outdated information about all processors' current capacity Based on this obsolete information, the anchor selects a receiver processor

- Distributed Data Balancer

  Another approach would be to leave the decision making to the individual processors i e a *distributed data balancer* Here, a processor wishing to download some of its nodes probes other processors for load information The probing can be done in two ways We assume that every processor has a list of participating processors in an array

  - Sequential Probing Here, processors begin probing other processors sequentially to share the load A processor will pick the first processor after itself and checks if that processor has sufficient capacity If not, it probes the next processor in line, and so on

  - Random Probing In this approach, the processors randomly probe other processors The randomly selected processor is checked for available capacity If it does not have enough capacity, then another processor (excluding the previously rejected ones) is selected randomly

After a receiver processor r has been selected, the sender s and the receiver r interact by a **negotiation protocol** (4.4). In this protocol, they decide exactly how many nodes are to be transferred from the sender s to the receiver r. The negotiation protocol is very essential since, in the interim that the receiver processor is selected and the actual node transfer takes place, the receiver or sender may experience more splits and hence a change in their capacities. Also, in the case of the centralized load balancing protocol, since the anchor has out of date information about each processor's status, the algorithm works well because of the negotiation protocol.

Performance

The performance of the dB-tree and the dE-tree depends on how the nodes are distributed among the processors, which in turn depends on the data balancing algorithm. In addition, data balancing incurs its own overhead.

There are many non-algorithmic factors that can affect performance. First, the number of hops that an operation requires to find its data increases with the height of the tree. Secondly, the width of replication increases with both increasing fanout and increasing numbers of processors that store the dB-tree. Finally, the manner in which additional storage is made available to the search structure affects the performance of the data balancing algorithm. To reduce the number of parameters we need to examine, our experiments used the following two scenarios:

- Incremental Growth

  When the storage for the distributed index runs low, the system manager must add storage capacity to some of the processors, or allow the dB tree to spread to more processors. Periodically, we perform incremental storage growth at the processors that store the dB tree. This is equivalent to adding a disk to a site or creating a new storage site. When a processor wishes to share some of its nodes, and all the currently active processors are near their threshold, a new processor may be started up or in the event that the processor limit is reached, a processor is selected randomly and its threshold is increased by a fraction of its current capacity The overloaded processor then shares its nodes with this new processor with newly added capacity

- Fixed Height Data Balancing

  To study the effect of large fanout on the width of replication, we fixed the height of the tree for all of the experiments.

To determine the nature of a large-scale dB-tree, we made a simulation study of data balancing on a dB tree We computed the number of message *hops* required to complete an operation, and the *width of replication*, or average number of copies of a node We are mainly concerned with the width of replication of level 2 nodes (which are most of the index nodes) The width of replication is a measure of the space overhead of maintaining a distributed index

Experiments, Results and Discussion **Experiment Description** We create an initial B-tree with a uniform random distribution of keys. After the initial B tree is created we vary the key distribution pattern dynamically To study the effect of our load balancing algorithm when the distribution changes we have introduced *hot spots* in our key generation pattern, where we concentrate the keys in a narrow range, thereby forcing about 40% of the messages to be processed at one or two 'hot' processors

To study the load variation behavior under execution, we collected distributed snapshots of the processors at intervals of every 10,000 keys inserted in the B tree At each snapshot we noted the processors capacity in terms of the number of leaves it has, the number of index level nodes, and the number of keys. We also noted the number of times a processor invokes the load balancing algorithm and the number of nodes it transfers.

Other important statistics are the number of message hops for a search, the width of replication and the number of probes required for load balancing. We also calculated the average number of times a leaf node moves between processors (taken with respect to the nodes in the entire B-tree).

To calculate the number of message hops for a search, we simulated 10,000 searches. A key to be searched is generated using a uniformly distributed random number. Since the path is replicated at each processor, every processor has a copy of the root of the tree. The search begins at the root of the tree on a randomly chosen processor. The search proceeds downward towards the leaves on the processor, and when a child has to be searched that is no longer on this processor, then a new random processor is chosen from among the processors that hold a copy of the child. This continues until a leaf node is reached. The message count is incremented each time a new processor is selected. We also noted at what level in the tree these processor boundaries are crossed. We finally calculated the average messages per search over all levels and over each level.

The width of replication indicates how widely the interior nodes are replicated in the distributed B tree. This gives us an idea of how the load balancing algorithms work. This also gives us an estimate of the number of message hops needed per search and also the amount of storage needed to store the B tree. If the algorithm does a good job at balancing the nodes at each processor, keeping logically adjacent nodes close then the number of copies of interior nodes is much less than having randomly scattered leaves at every processor, which makes every processor hold almost the entire index levels. We calculate the following two measures *average width of replication = # of copies of interior nodes / # of interior nodes*

A finer statistic is the width of replication at each level

average width of replication at level i = # of copies of level i nodes / # of level i nodes

We found that the width of replication is affected significantly by the choosing which leaf nodes to migrate. We first used random selection, where a processor that has to distribute its load chooses the leaf nodes randomly With this we found that the number of replicated copies was large. So, we improved upon this by sending out all leaves of a parent node. That is, we selected leaves sequentially The results obtained were much better and we present them below

Results

- Centralized and Distributed Data Balancing

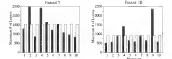


Figure 5 4 Performance of Load Balancing

The *Performance* bar charts (Figure 5 4) show the processors' capacity after the insertion of 100 000 keys When the "hot spots" distribution is used with node

Table 5 1 Load Balancing Statistics

Proce ssors	Average Number of Probes			Average Number of Moves		
	Centralized Load Balancing	Distributed LB (Sequential Probing)	Distributed LB (Random Probing)	Centralized Load Balancing	Distributed LB (Sequential Probing)	Distributed LB (Random Probing)
10	5 06	3.33	3 42	0 468	0 463	0 417
20	10 36	4.34	4 150	0 506	0 495	0 430
30	15 62	6 18	4 73	0 512	0 503	0 442
40	20 72	6 47	5 09	0 517	0 501	0 454
50	26 03	6 97	5 604	0 519	0.503	0 458

fanout 7, processors 2 and 4 are the *hot processors*, and receive a dispropor tionate number of inserts Without load balancing, the processors vary greatly in load with processors 2 and 4 having around 2500 leaf nodes and processor 3 having only around 800 leaves Our load balancing algorithm distributes the excess load at processors 2 and 4 among other processors, so that all processors contain about 1500 leaf nodes when all keys have been inserted With a node fanout average of 10, processor 9 stores an excess amount of leaves and the load balancing algorithm achieves a balance among all processors. The charts also show the reduction in storage as fanout is increased With a fanout of 7, all processors store about 1500 leaves (about 60% of the maximum storage), whereas with a fanout of 10, all processors store less than 1000 leaves (about 40% of the maximum storage)

Table 5 1 shows the calculated average number of probes made by the load bal ancing algorithm and the average number of moves made by a leaf node in the entire system The centralized load balancer requires a larger number of probes since the anchor does not know the exact status of all the other processors It has to handle stale information about the capacities of the processors Among the sequential probing and the random probing mechanisms, the random probing seems to require a little less number of probes than the sequential one In sequential probing, a processor (say 5) may be probed by two processors down the line (say 1 and 3) but can only serve one Hence, the second processor (3) may have to probe another processor before its request is met

The average number of moves of a leaf node show that on an average a leaf node moves only 5 times in the entire tree, so the load balancing overhead is not high

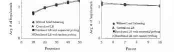


Figure 5 5 Average Number of Hops/Search

The graph of the *average number of hops/search vs processors* (Figure 5 5) shows that even as the number of processors increase, the hops/search does not increase linearly The number varying from 1 6 for 10 processors to 2 45 for 50 processors Even though the number of processors increases fivefold, the average number of hops increases less than twofold, thus indicating the good distribution of the nodes among processors

We also plotted the *average number of hops/search versus node fanout* and it is seen that as the fanout increases the number of hops decrease, as expected

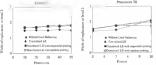


Figure 5 6 Width of Replication at Level 2

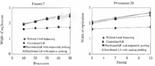


Figure 5 7 Width Of Replication

The *width of replication* charts (Figure 5 6, 5 7) show that in the case of no load-balancing the width of replication is lowest, while all the load balancing algorithms have about the same width of replication We have considered the width of replication at level 2 since most activity takes place at this level (one above the leaf) All the above graphs show that the width of replication is small, an average of about 2 Sequential selection of leaves has a lower width of replication than random selection as previously mentioned The width of replication over all levels is 4.9 for 30 processors and with node average fanout of 7 when leaves are selected randomly, while it is only 2.3 when leaves are selected sequentially. Similarly at level 2 it is 3.1 for random selection and 1.7 for sequential selection of leaves.

All the above results indicate our load balancing algorithms perform very well in maintaining a good and very close data balance among processors. The distributed random probing algorithm requires lesser number of probes and moves than our other algorithms. The distributed algorithm with sequential probing reduces the number of hops per search and width of replication more so than the others. Also, the sublinear increase indicates that the algorithms are suitable for scaling to large trees with large fanout and over many processors.

- Incremental Growth Data Balancing

  As explained above, in this algorithm, when none of the processors have available capacity, instead of increasing the capacity of every processor we select a processor randomly and increase its capacity. The results obtained show similar pattern to that of the general algorithms.

  The graphs in Figure 5.8 show that the average number of hops varies between 1.5 and 2.4 as we increase the number of processors from 10 to 50. The width of replication at level 2 (Figure 5.9) is about 1.7 and the width of replication over all levels (Figure 5.10) varies from 2 to 2.7

- Fixed Height Trees

  We performed simulations on fixed height large B-trees by inserting up to 2.5 million keys and varying the average fanout from 10 to 40 (average fanout is 69% of the maximum fanout [5]). In the first experiment we fixed the tree height

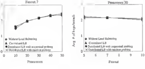


Figure 5.8 Incremental Growth Algorithm Average Number of Hops/Search

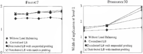


Figure 5.9 Incremental Growth Algorithm Width of Replication at Level 2

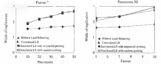


Figure 5 10 Incremental Growth Algorithm Width of Replication

to 4 When the root of the tree had the desired average fanout we collected statistics We noted the processors capacity in terms of the number of leaves it has the number of index level nodes, and the number of keys We also noted the number of times a processor invokes the load balancing algorithm, the number of probes required, the number of nodes that it transfers and the average number of times a leaf node moves between processors (taken with respect to the nodes in the entire B tree) The pattern of these statistics has been studied in the context of small fanout trees [30], so here we concentrate on other important statistics such as the average number of message hops for a search and the average width of replication at level 2

- Fixed height of 4 trees

We first performed experiments with fixed height trees of 4 The graphs in the Figures 5 11, 5 12, 5 13 5 14, 5 15 and 5 16 show the width of replication at level 2 and the width of replication over all levels, plotted against an increasing fanout for a fixed number of processors The graphs

17 through 5 19 show the variation of the number of hops with fanout for a fixed number of processors

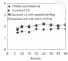


Figure 5 11 Height 4 Tree: Width of Replication at Level 2 for 10 Processors

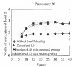


Figure 5 12 Height 4 Tree Width of Replication at Level 2 for 30 Processors

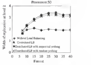


Figure 5 13 Height 4 Tree Width of Replication at Level 2 for 50 Processors

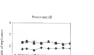


Figure 5 14 Height 4 Tree. Width of Replication for 10 Processors

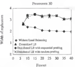


Figure 5 15 Height 4 Tree Width of Replication for 30 Processors

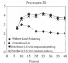


Figure 5 16 Height 4 Tree Width of Replication for 50 Processors

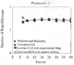


Figure 5 17 Height 4 Tree: Average Number of Hops/Search for 10 Processors

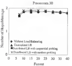


Figure 5 18. Height 4 Tree Average Number of Hops/Search for 30 Processors

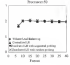


Figure 5 19 Height 4 Tree: Average Number of Hops/Search for 50 Processors

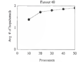


Figure 5 20 Height 4 Tree Variation of Average Number of Hops/Search with Processors

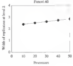


Figure 5 21 Height 4 Tree Variation of Width of Replication at level 2 with Processors

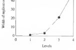


Figure 5 22 Height 4 Tree Variation of the Width of Replication with Level

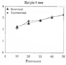


Figure 5 23 Height 4 Tree Linear Regression of the Width of Replicatio i

The WOR at level 2 reaches a plateau around 2.1 for 10 processors (5.11) around 2.8 for 30 processors (Figure 5.12) and 3.2 for 50 processors (Figure 5.13). Similarly the width of replication over all levels shows that for 10 processors the plateau is 2.24 (Figure 5.14) for 30 processors it is 3.1 (Figure 5.15) and for 50 processors it is 3.8 (Figure 5.16). We thus notice that the WOR at level 2 and the WOR over all levels reaches a plateau for a fixed number of processors as the fanout increases.

The number of hops required to perform an operation shows a similar phenomenon. Figures 5.17, 5.18, 5.19 plot the number of hops per operation against increasing fanout for a fixed number of processors. Again, the number of hops quickly reaches a plateau. From the table 5.2 we see that the number of hops is nearly constant with increasing fanout, and reaches a value of 1.99 for 50 processors.

For a comparitive study of the graphs in figures 5.11 through 5.19, we have condensed the data into a table 5.2.

From the table 5.2, we observe that for a dB-tree with a large fanout, the width of replication and the number of hops per operation depend on the number of processors only. Therefore we can predict the number of hops and the width of replication by studying the increase in the plateau value with an increasing number of processors.

- Number of Hops

  From the table 5.2 and figure 5.20 we see the effect of increasing the processors on the number of hops. Our results indicate that the hops do not increase significantly and reach only a value of 1.9. We conclude that in a large scale dB-tree with 4 levels, an average fanout

Table 5.8: Data for Fixed height of 4 dB-tree

Statistics	Processors	Fanout							
		7	10	15	20	25	30	35	40
WOR-2	10	1.74	2.02	1.87	2.00	2.25	2.16	2.06	2.1
	20	1.66	2.16	2.42	2.54	2.50	2.58	2.53	2.73
	30	1.67	2.37	2.81	2.78	2.88	2.9	2.75	2.8
	40	1.67	2.37	2.88	3.04	3.15	3.18	3.08	3.10
	50	1.67	2.37	2.89	3.23	3.34	3.45	3.23	3.23
WOR	10	2.32	2.44	2.25	2.25	2.42	2.3	2.16	2.24
	20	2.61	2.83	3.00	2.89	2.61	2.87	2.75	2.96
	30	2.62	3.41	3.47	3.31	3.29	3.34	3.07	3.1
	40	2.62	3.61	3.63	3.74	3.90	3.78	3.54	3.78
	50	2.62	3.73	4.17	4.07	4.24	4.08	3.78	3.8
Hops	10	1.35	1.41	1.46	1.44	1.47	1.44	1.40	1.38
	20	1.71	1.73	1.78	1.77	1.78	1.70	1.74	1.78
	30	1.87	1.85	1.84	1.84	1.88	1.90	1.81	1.8
	40	1.87	1.98	2.04	1.91	1.93	1.93	1.92	1.93
	50	1.87	2.02	2.04	1.87	2.02	1.96	1.98	1.98

of 40 and distributed over 50 processors, at most 2 hops per operation are required

- Width of Replication

In figure 5.21, we plot the plateau value of the width of replication at level 2 against the number of processors. The linear regression of the data shows that the slope for the random probing data is .0246. For the sequential probing algorithm, the slope is .0290. Based on the formula, width of replication at level 2 = 1.908 + .0248 * P we recalculated the width of replication and in figure 5.23 we show a comparison of the two set of values. We see that the experimental values are nearly those obtained theoretically. So, if we have a 1000 processors and a fanout of 1000, then the WOR for level 2 nodes is about 25 for random probing and 30 for sequential probing.

Another interesting characteristic is the variation of the width of replication with the level of the tree. In the path replication algorithm for the dB-tree, the width of replication for the root is the number of processors, and for the leaves is 1. We plotted the WOR for each level of the tree (figure 5.22), keeping the number of processors fixed (50) and fanout fixed at 40. The WOR at level 1 (leaf) is 1, while at 2 it is 3.2, at level 3 it is 23.3 and for level 4 it is 50. Thus, we see that in a 4 height tree the width of replication is less than half for the third level.

- Fixed height of 3 trees

For fixed height 3 trees, from the charts 5.24 through 5.32, we notice patterns similar to that of height 4 trees. We notice that the WOR reaches a plateau with increasing fanout for a fixed number of processors and the number of hops is nearly constant for a fixed number of processors. However, the WOR at level 2 is higher, reaching a value of 6.71 for 50 processors. The WOR over all levels is 7.74 for 50 processors. For the height 4 tree, the WOR at level 2 is 3.2 and for level 3 (at one level lower than the root) it is 23.3, whereas here the WOR at level 2 (one level lower than the root) is 6.71 and the maximum number of hops was 1.69. Here too, we took a linear regression of the variation of the width of replication at level 2 with the number of processors and obtained a formula for the width of replication at level 2 as $2.795 + .0815 * P$. We recalculated the width of replication at level 2 using this formula and in figure 5.33 we show the experimental and theoretical values obtained. Again, we can conclude that the WOR and the number of hops are greatly affected by the number of processors over which the B tree is distributed.

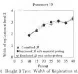


Figure 5.24 Height 3 Tree: Width of Replication at Level 2 for 10 Processors

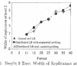


Figure 5.25 Height 3 Tree: Width of Replication at Level 2 for 30 Processors

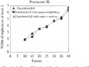


Figure 5 26 Height 3 Tree Width of Replication at Level 2 for 50 Processor

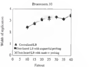


Figure 5 27 Height 3 Tree Width of Replication for 10 Processors

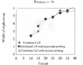


Figure 5 28 Height 3 Tree Width of Replication for 30 Processors

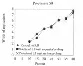


Figure 5 29 Height 3 Tree: Width of Replication for 50 Processors

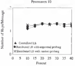


Figure 5 30 Height 3 Tree Average Number of Hops/Search for 10 Processors

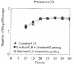


Figure 5 31 Height 3 Tree Average Number of Hops/Search for 30 Processors

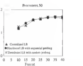


Figure 5 32 Height 3 Tree Average Number of Hops/Search for 50 Processors

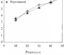


Figure 5 33 Height 3 Tree Linear Regression of the Width of Replication

Fixed height of 5 trees

For fixed height trees of height 5, we could not get statistics for fanouts larger than 20 as the tree was very big. The WOR at level 2 is 2.27 for 50 processors and the WOR over all levels is 2.64 for 50 processors. The number of hops is maximum at 2.21. We include the charts 5.34 through 5.42 and table 5.4 for the sake of completeness.

Table 5.3 Data for Fixed height of 3 dB tree

Statistics	Processors	Fanout						
		10	15	20	25	30	35	40
WOR 1	10	1.92	2.31	2.76	3.04	3.13	2.97	3.27
	20	1.92	2.81	2.81	3.60	4.56	4.08	4.66
	30	1.92	2.81	3.52	3.92	4.01	5.03	5.03
	40	1.92	2.81	2.81	3.85	4.97	4.84	5.93
	50	1.92	2.81	3.52	3.85	4.74	5.23	6.71
WOR	10	2.46	2.76	3.09	3.30	3.14	3.17	3.43
	20	2.46	3.71	3.71	4.30	5.03	4.8	5.02
	30	2.46	3.71	4.94	4.89	5.41	5.72	6.21
	40	2.46	3.71	3.71	4.03	6.06	5.92	6.74
	50	2.46	3.71	4.64	4.02	6.16	6.53	7.74
Hops	10	1.10	1.27	1.33	1.32	1.33	1.29	1.28
	20	1.19	1.43	1.43	1.53	1.64	1.56	1.56
	30	1.19	1.43	1.58	1.58	1.60	1.62	1.59
	40	1.10	1.43	1.43	1.6	1.68	1.63	1.67
	50	1.10	1.43	1.56	1.6	1.68	1.70	1.80

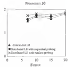


Figure 5 34 Height 5 Tree. Width of Replication at Level 2 for 10 Processors

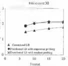


Figure 5 35 Height 5 Tree: Width of Replication at Level 2 for 30 Processors

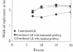


Figure 5 36 Height 5 Tree: Width of Replication at Level 2 for 50 Processors

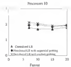


Figure 5 37 Height 5 Tree Width of Replication for 10 Processors

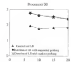


Figure 5 38 Height 5 Tree Width of Replication for 30 Processors

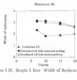


Figure 5 39. Height 5 Tree Width of Replication for 50 Processors

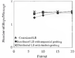


Figure 5 40 Height 5 Tree: Average Number of Hops/Search for 10 Processors

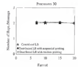


Figure 5.41 Height 5 Tree Average Number of Hops/Search for 30 Processors

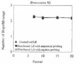


Figure 5.42 Height 5 Tree Average Number of Hops/Search for 50 Processors

Table 5.4 Data for Fixed-height of 5 dB tree

Statistics	Processors	Fanout			
		7	10	15	20
WOR-2	10	1.57	1.73	1.69	1.83
	20	1.82	1.99	2.02	2.02
	30	1.88	2.05	2.10	2.15
	40	2.10	2.25	2.01	2.21
	50	1.90	2.25	2.30	2.27
WOR	10	1.96	1.98	1.85	1.92
	20	2.34	2.28	2.19	2.18
	30	2.77	2.63	2.30	2.38
	40	2.93	2.78	2.51	2.51
	50	3.12	3.15	2.98	2.64
Hops	10	1.47	1.55	1.51	1.57
	20	1.87	1.82	1.93	1.63
	30	2.01	2.07	2.03	2.01
	40	2.15	2.11	2.13	2.13
	50	2.28	2.21	2.23	2.21

Table 5.5 Comparison of Fixed height 3, 4 and 5 trees with Fanout 20 and over 50 processors

Height	WOR at level 2	WOR	Number of Hops
3	3.52	4.64	1.88
4	3.23	4.07	1.97
5	2.27	2.64	2.21

To conclude we observe that the results obtained favor the scalability of the distributed B-tree The number of hops depends on the height of the tree and the WOR depends on the number of processors, but grows very slowly From the comparitive table for different height trees shown in table 5.3 we can see how the width of replication and average number of hops per operation vary with tree height

5.3.2 The dE-tree

The dE-tree is a practical distributed index constructed from the distributed B-tree. The main purpose of this being to reduce the communication cost by storing fewer leaves and thus incuring less overhead We have observed that, instead of maintaining separate leaves for the consecutive keys stored in a processor, a more effective approach is to maintain a single leaf (i e, an *extent*) that maintains key range information only, and store the keys in a local data structure.

The difference between the dB tree and the dE-tree is that it is the load balancer that decides whether to split or merge a leaf node. The load balancer is invoked when a key is inserted into a leaf node If the load balancer decides that the processor holds too many keys it decides to download some of its keys to some other processor It selects a leaf node and decides to either perform a merge or a split The processor with which to merge or give away the split sibling is also selected based on certain criteria. We will explain the selection criteria for the leaves and the processors below

Algorithms

In each of these algorithms, the load balancer decides if the processor has an excess number of keys Let the excess number of keys be k

Random. As the name suggests the leaf node to be merged or split is selected randomly

- Step 1 Pick a random node If that node is owned by processor **P** then go to step 2 otherwise go to step 1
- Step 2 If the node **n** has a right neighbor **r** and **r**'s owner processor has available capacity, then transfer the excess nodes and stop
- Step 3 If the left neighbor **l** of **n** has available capacity, then transfer the excess nodes and stop

<u>Merge.</u> Here, we select a leaf node such that it can be merged with either its left or right neighbor If there is no such leaf node, then the largest extent leaf is chosen for a split

- Step 1 Scan through the list of nodes until an extent is found that is owned by the processor **P** that has at least **K** keys Let this node be say **n**.
- Step 2 If the node **n** has a right neighbor **r** and **r**'s owner processor has available capacity then transfer the excess nodes and stop
- Step 3 If the left neighbor **l** of **n** has available capacity, then transfer the excess nodes and stop
- Step 4 If you cannot find a processor that can take all the excess keys **K**, then scan through the list of nodes till you find another node owned by this processor **P** with the largest number of keys If found, go to step 2, else continue (If a null node is reached indicates all nodes have been searched)
- Step 5 Try and merge the keys of node n with either its right or left neighbor (as in step 2 or step 3) If neither of them can take the keys, select a processor say **R** randomly and increase its capacity This is equivalent to adding extra disk space at one particular processor

- Step 6 See if **R** is either the right neighbor's owner or the left neighbor's owner If so, merge with the right neighbor or left neighbor by transferring the excess keys and stop

- Step 8 Split the node n Give the new sibling to processor **R** Stop

Aggressive Merge. In the above merge algorithm, we search through the set of nodes owned by the processor for one such that a neighbor can take all keys offered In the aggressive merge approach we first search for a node that the processor owns such that a neighbor can take all of the keys in the node Then if we cannot find any neighbor that can take all keys, we settle for sending lesser (than **k**) number of keys. So we search for a neighbor that can take the most number of keys less than **k** The strategy works because on the next insert the processor will balance again

- Step a Set **merge_node** = NULL, Set **maximum** = 0,

- Step b Pick the first node on the list **n** owned by the processor **P** that has excess keys **k** Let the right neighbor processor **R** have free space **f** If (f > **maximum**) set **merge_node** = **n** Merge with the neighbor by transferring the minimum of **f** and **k** Stop

- Step c Scan through the list and pick the next node n in line. If the end of the list is reached, go to step d Let the right neighbor have free space, **free** If **free** > **maximum** set **merge_node** = n and **maximum** = **free** Go to step c

  Step d If **maximum** is 0, then go to step 6 of the merge algorithm Else **merge_node** gives the node that can be merged with its right neighbor by giving away **maximum** keys. Stop

Experiments, Results and Discussion

The simulation of the dE-tree is similar to that of the dB tree, except that the leaves hold key ranges (extents) and can hold an arbitrary number of keys The interior nodes have an average fanout defined as 70% of the maximum fanout We performed experiments with average fanouts of 5, 7 and 10 A total of 500 000 keys were inserted and upto 50 processors were used for distributing the B tree

Experimental Setup. A uniform random distribution of keys is chosen to create the initial dE tree Initially each processor was given one leaf node with a range of keys

To study the load variation behavior under execution, we collected distributed snapshots of the processors at intervals of every 50,000 keys inserted in the dE-tree. At each snapshot, we noted the processors capacity in terms of the number of leaves it has, the number of index level nodes, the number of keys the number of splits, merges, and deletes. We also noted the number of times a processor invokes the load balancing algorithm and the number of nodes that it transfers.

Similar to the dB tree other important statistics are the number of message hops for a search, the width of replication and the number of probes required for load balancing We also calculated the average number of times a leaf node moves between processors

Results. We first compared the *random* and the merge algorithms In this experiment we built a dE-tree of an average fanout of 10 in the interior nodes, with 500,000 keys and used from 10 to 50 processors We observed the two algorithms behaved quite similarly for certain statistics We noticed both the algorithms did a good job at maintaining a data balance with the mean being around 74 and the variance being 0 000001 The number of hops per message also varies similarly in both algorithms, from 1 18 to 2 04 while varying the processors from 10 to 50 in a tree of height 3 The width of replication varied between 5 8 to 7 13 for the algorithms

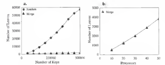


Figure 5 43 dE-tree Comparison of the Random vs Merge Algorithms

The difference in the algorithms is reflected in the number of leaves and the number of interior nodes that are stored at each processor We see from the graph 5 43a that a dE-tree distributed over 30 processors the random algorithm stores 56000 leaves whereas the merge algorithm stores around 2000 leaves This shows that the merge algorithm does a far superior job at reducing the storage overhead of the dE-tree However, the number of merges that occur is about 1000 for the random algorithm whereas for the merge algorithm the number is 1900 Also, there is a restructuring penalty for the merge algorithm with 70 nodes and 346 copies being touched (i e involved in the restructuring) while only 16 nodes and 71 copies are touched with the random algorithm

The results obtained from the above algorithms indicate definitely that the merge algorithm is more efficient in reducing storage space without affecting the number of hops per message and the width of replication So we decided to explore the

Table 5 6 Merge Algorithm Comparison of dE-trees with 2.5 Million and 5 Million Keys

Keys	Processors	Leaves	Interior Nodes	Nodes touched	Copies touched	Splits	Merges
2.5 Million	10	508	54	115	623	547	1971
	20	1235	129	104	733	1362	3946
	30	2043	218	103	990	2301	5932
	40	2926	303	106	1328	3413	7902
	50	3847	412	103	1330	4605	9761
5 Million	10	516	56	108	568	551	6735
	20	1235	129	104	718	1361	4991
	30	2048	213	103	994	2329	5924
	40	2940	301	107	1344	3413	7880
	50	3811	400	106	1380	4533	9861

merge algorithm further One of the studies was to see the effect of the number of keys in the dE-tree on the number of leaves and interior nodes So we performed experiments with 2 5 million and 5 million keys

From the Table 5 6 we see that for 10 processors, in a dE-tree with 2 5 million keys, the number of leaves is 508, interior nodes is 54, the number of nodes touched for restructuring is 115 and copies 623 The corresponding numbers for a dE-tree with 5 million keys are, the number of leaves 516, interior nodes 56, nodes touched 108 and copies 568 So we see that increasing the number of keys did not greatly increase the number of leaves and interior nodes The same pattern can be seen for 20, 30 40 and 50 processors

It can also be seen that both the number of leaves and interior nodes increase nearly linearly with the number of processors (figures 5 44, 5 45, 5 46 and 5 47)

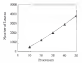


Figure 5 44 Effect of Increasing the Number of Processors on the Number of Leaves stored in a dE-tree with 2.5 million Keys for the Merge Algorithm

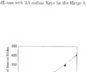


Figure 5 45 Effect of Increasing the Number of Processors on the Number of Interior Nodes stored in a dE tree with 2 5 million Keys for the Merge Algorithm

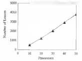


Figure 5.46 Effect of Increasing the Number of Processors on the Number of Leaves stored in a dE-tree with 5 million Keys for the Merge Algorithm

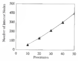


Figure 5.47 Effect of Increasing the Number of Processors on the Number of Interior Nodes stored in a dE-tree with 5 million Keys for the Merge Algorithm

Another observation is that the number of copies touched for restructuring is more or less constant as the number of processors increase.

Thus far we observed that the merge algorithm reduces the number of leaves and interior nodes, and increasing the size of the dE-tree does not have a great effect on the number of leaves and interior nodes So, the next step was to see if we could reduce the number of leaves even further by varying some input parameters to the experiment.

<u>Extensions.</u> The main objective of performing further experiments is to observe the effect of the input parameters on the number of leaves in the dE-tree

- Questions to be answered
  - What are the input parameters that effect the extents of the dE-tree?
  - How does one vary the selected input paramters?

- Experiment. In our original experiment, we inserted 2 5 million keys in total in a dE-tree with an average fanout of 10 We built our initial dE-tree by assigning some number of keys to each processor After the initial dE-tree is built, keys are inserted for the dE-tree to grow When a processor decides that it hold too many keys it invokes the load balancer, which attempts to distribute the keys among the active processors If none of the active processors have available capacity, then a processor is chosen and its capacity increased by an increment We noticed that by selecting the number of keys to build the initial dE-tree and the increment appropriately, we could reduce the number of extents in the final dE-tree Thus, we chose the *Initial_keys* (keys to build a small initial dE-tree) and the *Increment* which is the storage added to a processor during load

Table 5 7 Comparison of Doubling Initial Keys and Increment for a dE tree with 2 5 Million Keys

Algo	Processors	Leaves	Interior Nodes	Nodes touched	Copies touched	Splits	Merges
Double	10	535	53	115	554	567	7303
Initial Keys	20	1234	126	106	726	1337	3927
Original	30	2040	223	96	931	2251	5892
Increment	40	2897	307	101	1185	3285	7814
	50	3817	396	105	1363	4446	9740
Original	10	283	31	87	409	299	900
Initial Keys	20	688	70	93	516	754	1906
Double	30	1178	125	80	612	1359	2631
Increment	40	1719	189	71	735	2028	3745
	50	2284	241	76	913	2818	4553

balancing, as the two input parameters to vary The increment remains the same for the entire run

The next concern is how to vary these paramters We started off by allowing a growth of 50 times for the dE-tree, hence if the final dE-tree holds 2 5 million keys, then *Initial_keys* is chosen as 2 5$million/(50*numberofprocessors)$ The increment is chosen as $2*initial\_keys$

- Various Scenarios and their Results

  We have the following scenarios

  - *Double Initial_keys Original Increment*

    In this scenario we allowed a growth of the tree 25 times so we inserted (2 5$million/50*numberofprocessors$)/2 keys initially The increment was chosen as 2 * 2 5$/(50*numberofprocessors)$ The number of leaves stored at the end of the run was 535 and the number of interior nodes was 53 The number of nodes touched for restructuring was 115 and copies 554 (for 10 processors) (Table 5 7)

- *Original Initial_keys, Double Increment*

  Here, we allowed a growth of 50 times for the tree and hence inserted $2\ 5million/50 * numberofprocessors$ and doubled the increment to $2*(2*2\ 5million/(50*numberofprocessors))$ The number of leaves stored at the end of the run was 283 with interior nodes being 31 Nodes touched for restructuring was 299 and copies 409 (Table 5 7)

The numbers obtained above show that the *initial_keys same, double increment* method reduces the number of leaves nearly by half Comparing this to the original algorithm, with $initial\_keys = 2\ 5/(50 * numberofprocessors)$ and $increment = 2 * 2.5/(50 * numberofprocessors)$ we see that the number of leaves has reduced from 508 to 283 and interior nodes from 54 to 31

The results obtained from performing the experiments were interesting enough to prompt us to explore the effect of the variation of *Initial_keys* and *Increment* further We noticed that it was the *Increment* that was added to a processor that affected the number of leaves in the dE-tree Hence, we varied the *Increment* keeping the *Original Initial_keys*

After performing the experiments with these two scenarios to investigate further we came up with three other scenarios We added the following scenarios (as listed in table 5 8)

- *Original Initial_keys Half Increment*
- *Original Initial_keys Quarter Increment*

Table 5.8 Various Scenarios of the Input Parameters for a dE-tree of 2.5 Million Keys

Scenario	Number of Keys	Increment
Original Keys Original Increment	K	I
Original Keys Double Increment	K	2*I
Keys doubled Original Increment	2*K	I
Original Keys Half Increment	K	I/2
Original Keys Quarter Increment	K	I/4
Original Keys Tenth of Increment	K	I/10
K = 2.5/50 * (number of processors)		
I = 2 * K		

Table 5 9 Effect of Changing the Increment on a dE tree with 2 5 Million Keys

Increment	Processors	Leaves	Interior Nodes	Nodes touched	Copies touched	Splits	Merges
2*Increment	10	283	31	87	409	299	909
Increment	10	508	54	115	623	547	1971
Increment/2	10	962	103	128	719	1050	18681
Increment/4	10	1892	205	136	954	2121	19183
Increment/10	10	9098	953	179	1265	10487	66792

– *Original Initial_keys Tenth Increment*

We observed that halving the increment increased the number of leaves to 962, reducing the increment to a quarter of its original brought the number of leaves to 1892 and interior nodes to 205 Thus, changing the increment from 5 to 1 and then to 2, changed the number of leaves from 962 to 508 to 283 (table 5 9) showing an almost linear dependence of the number of leaves on the increment added to a processor during load balancing From this we can conclude that the size of the tree depends on the number of times the increment is performed With a small increment the number of times the increment is performed is large and so also, the number of leaves

The above discussion has shown that the merge algorithm and its various scenarios definitely show an improvement over the random algorithm The next step was to see if we could improve the results even further by designing a new algorithm so we developed the aggressive merge algorithm We show a comparison of the merge and the aggressive merge algorithm in figure 5 46a for

30 processors The number of leaves for the merge algorithm are about 2048, whereas for the aggressive merge the number of leaves is only 338 The plot shows us that the aggressive merge algorithm is definitely more efficient

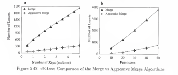


Figure 5 48 dE-tree: Comparison of the Merge vs Aggressive Merge Algorithms

We also plot the number of leaves versus processors in Figures 5 43b and 5 48b and note the quadratic behavior of the curves So, the aggressive merge algorithm does a much better job at reducing the storage overhead at each processor, while increasing the cost of restructuring as expected

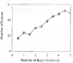


Figure 5 49 dE-tree Number of Leaves versus Keys for 10 processors for Aggressive Merge Algorithm

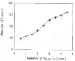


Figure 5 50 dE-tree Number of Leaves versus Keys for 20 processors for Aggressive Merge Algorithm

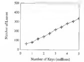


Figure 5 51 dE-tree Number of Leaves versus Keys for 30 processors for Aggressive Merge Algorithm

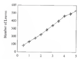


Figure 5 52 dE-tree Number of Leaves versus Keys for 40 processors for Aggressive Merge Algorithm

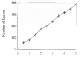


Figure 5 53 dE-tree: Number of Leaves versus Keys for 50 processors for Aggressive Merge Algorithm

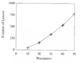


Figure 5 54 dE-tree. Number of Leaves versus Processors for Aggressive Merge Algorithm

In figures 5.49 through 5.53, we plot the number of leaves versus the number of keys for different numbers of processors, varying them between 10 and 50, for the aggressive merge algorithm. We also plot the number of leaves versus processors for 5 million keys and note the quadratic nature of the curve (Figure 5.54). It can be seen from the charts (5.49 and 5.50) that the number of leaves is flattening out, reaching a plateau for the plot of 10 and 20 processors. A good algorithm should have no more than about n(n-1)/2 leaf nodes (a processor is neighbors with every other one). Our aggressive merge algorithm achieves this as the number of leaves flattens out with increasing numbers of keys for 10 and 20 processors. For 30 or more processors the simulation did not execute long enough to reach a plateau value, as the final number of leaves is less than n(n-1)/2 for $n >= 30$.

As for the dB tree algorithms, here too we observed the width of replication at all levels, the height of the tree and the number of hops per message for a dE-tree with 5 million keys. We see that the height of the dE-tree is 3 for 10 processors and 4 for 20 to 50 processors, with the number of hops varying from 1.05 to 1.74 as we increase the processors from 10 to 50. The width of replication at level 2 varies between from 6.14 to 10.65. We thus see that our algorithm does not significantly increase the space and message overhead.

All the above observations lead us to conclude that of all the algorithms, the aggressive merge algorithm performs the best, having far fewer nodes in the dE-tree.

5.4 Timing

This chapter has thus far concentrated on the performance of replication and balancing algorithms from a qualitative point of view, by doing large scale simulations. Here, we are concerned with characteristics such as system response times and throughput. The timing information gives us an idea of how fast the system responds to a query and what the throughput of the system [illegible] in terms of the number of queries it can process per second To obtain these timings, we go back to the implementation of our distributed B tree that we discussed in Chapter 4.

5.4.1 System Response Time

Response time means the time taken for a single query to be processed The anchor process sends out a operation query and waits till an answer comes back from a node manager that the operation has been completed The total time taken for the operation to complete is noted The anchor then sends out the next query The average of the time taken for all operations gives the response time for a single operation. Response time is defined as

response time = total time taken for all queries / number of queries

Generation rate is defined as

generation rate = total number of messages generated / time taken to generate the messages

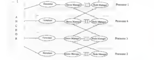


Figure 5.55 Experimental Model for Measuring System Throughput

Experiment. Each processor has a separate process, the *generator* that generates the messages (Figure 5.55). The generation of the messages is governed by the anchor, that gives a slot of time during which the generators generate messages. The generator communicates with the queue manager by a socket connection. It sends the time-stamped messages to the queue manager at that processor which queues the messages in the message queue for the node manager to pick up. The message travels to the correct leaf and once the operation is completed, the node manager time-stamps the message and returns it to the anchor. The anchor thus obtains the time taken for each message. After it receives all the messages, it then calculates the average response time and generation rate.

We have chosen to observe the response times of 4, 6, and 8 processors. In our experiments, we obtained different generation rates by varying the sleep interval between consecutive messages and noted the response times. Our experiments show that the response time decreases as the generation rate increases. This could be because of better access to the CPU, fewer page misses and/or cache misses. After a certain generation rate the response time is expected to increase as the system is driven to its limit and begins to slow down due to queueing delays, etc. We have not been able to observe this trend in our experiments, even though we employed the maximum generation rate possible. The generation rate is limited by the system clock granularity and thus from our experiments, we observe that the generation rate possible (with no sleep intervals) is not sufficiently large enough to flood the system with messages. So, for all practical purposes, we can safely assume that we need to consider the lowest response time as our system response time.

Results. The graphs show the response times and generation rates for 4, 6 and 8 processors.

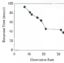


Figure 5 56 Response Times for a 4 Processor System

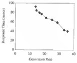


Figure 5 57 Response Times for a 6 Processor System

It was observed that for a 4 processor system, it takes about 35 milliseconds for an operation to complete, for a 6 processor system it takes 40 milliseconds and for an 8 processor system, the response time is 55 milliseconds

In order to justify these timings, it is necessary to know the message transit time, processing time at a processor and queuing time It is difficult to get an estimate of the queuing time, but we performed some simple experiments to determine the message transit time and the processing time

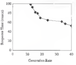


Figure 5 58 Response Times for a 8 Processor System

Experimental Model. For the processing time at any processor *unit_processing_time*, we use the same model that we used to collect timings When the node manager receives a message it time stamps it with the processing_start time and after processing the message, it again time-stamps the message with the processing_end time All the messages are returned to the anchor, so the anchor calculates the average *unit_processing_time*.

To calculate the message travel time between any two processors *unit_message time*, the anchor spawns off processes on different machines and messages are sent back and forth between all the processes and the anchor Each message is time-stamped with the time that it was sent and the time it was received at another processor The anchor then finally collects all the messages and calculates the average time it takes for a message to travel between any two processors

From our experiments we observed that the message transit time between two processors is approximately 5.2 milliseconds and the unit processing time is around 4 4 milliseconds. In table 5 10 we calculate the processing times and message transit

Table 5 10 Timing Calculations

Processors	Number of Hops	Processing Time	Message Transit Time	Total Time	Time Difference
4	1 6	(1 6+1)*4.4 = 11 44	1 6*5 2 + 2*5 2 =18 72	30 16	35-30 16 4 74
6	1 9	(1 9+1) * 4 4 =12 76	1 9*5 2 + 2*5 2 =20 28	33.04	40-33 04 =6 66
8	2.3	(2 3+1)*4 4 =14 52	2 3*5 2 + 2*5 2 =22.36	36 88	50-36.88 =13 12
Unit_processing time = 4 4 milliseconds					
Unit_message_time = 5 2 milliseconds					

ture as follows.

processing time = (number of hops + 1) * unit_processing_time.

message transit time = (number of hops * unit_message_time) + roundtrip time to anchor

The roundtrip time to anchor is added since a message starts at the anchor and returns to the anchor

The time difference in the table 5 10 can be attributed to the delays that include message collisions process context switches disk swaps etc

5.5 Performance Model

In this section, we present a simple analytical model that predicts operation response times and the maximum throughput of the distributed search structures described in this paper The performance depends on the structure of the dB tree or dE-tree For example, both the number of hops per operation and the degree of replication affect the amount of overhead required to maintain the search structure These values are very difficult to calculate, and they depend on the algorithm used to perform the data balancing For this reason, we will use the estimates of the number of hops and the degree of replication developed in Section 5 3 1 The model described in this section is loosely based on the model presented in [28] We assume that operations are generated uniformly at all processors, and the accesses are made to the data uniformly

We first define the variables that we use in the analysis

L Number of levels in the search structure (level 1 is the leaf, level L is the root)

P Number of processors that maintain the search structure

H Average number of hops required to navigate to a leaf

R_i Degree of replication at level i, $i = 1, \ldots, L$ $R_1 = 1$ and $R_L = P$

F Maximum node fanout

q_i Probability that an operation is an insert operation

p_{res} Probability that an operation causes restructuring (split or merge)

t_s Message transmission time

t_a Time to process an action

t_m Processing time for sending and receiving a message

λ Arrival rate of operations to a processor

λ_{tot} Total arrival rate of operations to the distributed search structure

N_a Average number of actions generated by an operation

N_m Average number of messages generated by an operation

W Waiting time

T Response time of an operation

Th_{max} Maximum throughput

We start by determining the number of messages and actions required to process an operation, N_a and N_m. Since there are L levels, L search actions are required. Since each operation requires H hops, $H+1$ messages are required (a slightly pessimistic estimate). In addition, an operation might cause restructuring. If there are more inserts than deletes, then $p_{res} \approx 1/(.68 * F)$ [28]. When a node splits, the sibling is created, its right and left neighbors must be informed, and all copies of the parent must be informed about the new sibling. In turn the parent might split, with probability p_{res}. Therefore

$$N_a = L + q_u \sum_{i=1}^{L-1} p_{res}^i (3R_i + R_{i+1}) \tag{5.1}$$

$$N_m = H + q_u \sum_{i=1}^{L-1} p_{res}^i (2R_i + R_{i+1} - 1) + 1 \tag{5.2}$$

If λ is the rate at which operations are generated at a node that helps to maintain the distributed search structure, then the total rate at which operations are generated is

$$\lambda_{tot} = P\lambda \tag{5.3}$$

A processor that helps to maintain the distributed search structure will be required to process jobs that correspond to actions and jobs that correspond to message passing. The average time to process a job is

$$t_{avg} = (N_a t_a + N_m t_m)/(N_a + N_m) \tag{5.4}$$

Since the root is fully replicated, it is not a bottleneck. If the data balancing distributes the nodes properly, then no leaf node is a bottleneck either. Therefore, the work to execute an operation is evenly spread among the processors in the system. As a result, the processor utilization due to search structure processing is

$$\rho = \lambda/(N_a t_a + N_m t_m) \tag{5.5}$$

The time that a job spends waiting for processor service can now be calculated by applying a queuing model. We use a simple M/M/1 queue, and find that

$$W = t_{avg}\frac{\rho}{1-\rho} \tag{5.6}$$

The time to get a response from an operation is the time to process all messages and actions associated with the operation

$$T = L(W + t_a) + (H + 1)(W + t_a + t_m) \tag{5.7}$$

The maximum throughput is the maximum rate at which every processor can execute the jobs associated with the search structure operations

$$Th_{max} = P/(N_a t_a + N_m t_m) \tag{5.8}$$

In a distributed search structure with a large number of processors, the overhead of maintaining the search structure is primarily due to the number of hops, H, and the cost of maintaining the level 2 nodes. As we saw in Section 5.3.1, H approaches an asymptote for a fixed height tree. The algorithms described in [26] require R_2 actions for every split of a level 1 node. Fortunately, we found that R_2 grows very slowly with increasing P. As a result, the overhead of maintaining a dB-tree does not increase as fast as the processing power of the system increases when processors are added. As result, the dB tree algorithm is scalable to a very large number of processors.

5.5.1 An Application

- Analysis for a 8 processor dB tree:

  Let us make an analysis of a dB-tree distributed over 8 processors for which we have performed the timing experiments So, we have $P = 8$ and average fanout $f = 10$ In Section 5 3 1, we saw that in a large-fanout dB tree with 4 levels, the number of hops is about 2, and the width of replication on level 2 is about 1 508 + 0248 $* P$, where P is the number of processors We have found that the level 3 nodes are replicated at nearly half the number of processors, so we will assume that $R_3 = P/2$ We measured the time to process a message as $t_a =$ 0044 seconds and transmission time for a message as $t_s =$ 0052 seconds (5 10)

  With these statistics in mind, we will use the following additional parameters as input to the model

$$t_m = 001$$

$$q_i = 1$$

$$y_{rep} = 1/(f) = 1$$

  We use these parameters to determine the number of messages and actions that an operation generates

$$N_a = 4\ 063$$

$$N_m = 3\ 040$$

  We can use the the estimates of the number of actions and messages to compute the average execution time and the maximum throughput

$$t_{avg} = 0028$$

$$Th_{max} = 382$$

With a processing rate of 191 operations per second $\rho = 1/2$ and the response time for an operation is 0565 seconds From the chart 5 58, we see that the lowest response time is around 050 seconds Our analysis, gives a more pessimistic value taking into account some queuing time.

- Analysis for a 50 processor dB tree

We will do a similar analysis for larger dB trees that we used in our simulations in section 5 3 1 with $P = 50$ and average fanout $f = 40$ From our experiments we obtained the Widths of Replication at level 1 as $R_1 = 1$, at level 2 $R_2 = 3\ 2$, at level 3, $R_3 = 23\ 3$ and at level 4, $R_4 = 50$ (5.22) The number of hops is 2 for a large dB-tree with 4 levels

Again, similar to the analysis for 8 processors, here too we use additional parameters as input to the model:

$$t_a = \quad 0044$$

$$t_s = \quad 0052$$

$$t_m = \quad 001$$

$$q_i = \quad 1$$

$$p_{res} = \; 1/(f) = \; 025$$

We then compute the number of messages and actions that an operation generates

$$N_a = 4\ 918$$

$$N_m = 3.013$$

We now calculate the average execution time and maximum throughput

$$t_{avg} = 0025$$

$$Th_{max} = 2417$$

With a processing rate of 1209 operations per second, $\rho = 1/2$, and the response time for an operation is 0565 seconds

For a comparison, consider the performance of a centralized index server that has the same message passing cost, $t_m = 001$ Servicing each request requires the processing of two messages (the request and the response) We will assume that the actual index lookup requires $t_s = 0044$ seconds Then, servicing an operation requires 0064 seconds, allowing a maximum throughput of 156 25 operations per second If the processing rate is 78 operations per second, then the response time for an operation is 030 seconds Therefore, at the cost of doubled latency, the throughput is increased by a factor of 15 by using the distributed search structure.

5.6 Conclusion

In this chapter, we have described extensively all the algorithms developed, experiments conducted and the performance results obtained Here, a brief summary is presented by listing the conclusions drawn from our experiments

- Replication In section 5 2 we have presented two algorithms for replication namely *full replication* and *path replication* and discussed the method of maintaining replica coherency To compare the performance of the algorithms we examined the overhead of the full and path replication of the index nodes and found that path replication imposes much less overhead in terms of space and messages than full replication The width of replication measure for path replication shows a sublinear increase with number of processors and hence permits a scalable distributed B tree

- Data Balancing We also conducted simulations on a large scale to validate the results obtained from our implementation The simulation results show that our algorithms for data-load balancing achieve a good data balance among processors without imposing much overhead. An average node moves only about 5 times in the entire tree so the load balancing overhead is not high The centralized and distributed data balancers perform equally well, with the distributed algorithm using sequential probing achieving a good balance keeping the width of replication small, on an average of 2.

  We varied certain parameters to our simulation and performed experiments on two scenarios:

  - Incremental Growth Data Balancing The results of this experiment are similar to those of the general algorithms with the width of replication being 1 7 and the number of hops around 2 4 for 50 processors
  - Fixed Height Data Balancing We performed experiments with fixed-height trees of 3, 4 and 5, with fanouts varying from 10 to 40 and the number of processors varying from 10 to 50 With all of them we noticed that the width of replication reaches a plateau with increasing fanout. For example, from table 5 2 we see that with a fanout of 40 and with 50 processors, in a tree of height 4 the width of replication at level 2 is 3 23 and the width of replication over all levels is 3 8 and the number of hops is 1 89 This is in accordance with the formula we have derived for the width of replication at level 2 which is

width of replication at level $2 = 1.908 + .0348*P$, where P is the number of processors.

We also notice that the width of replication and number of hops depend only on the number of processors. The fixed height dB tree experiments show that our algorithms are suitable for larger trees with a large fanout. Thus, all our algorithms make the B-tree scalable.

- dE-tree: We also designed the distributed extent tree that is data balanced on the number of keys held by a processor. We first compared two algorithms, random and merge. Of these we found that the merge algorithm reduced the number of extents in the dE-tree. Increasing the size of the dE-tree did not overly reduce the number of extents even with the merge algorithm, so we made some extensions by changing the input parameters to the dE-tree. We varied the initial number of keys and the increment added to a processor when it runs short. We found that the number of extents varies linearly with the number of times the increment is performed, with the number of leaves being large when the increment is small (table 5.9). We then developed the aggressive merge algorithm, where we settle for sending as many keys as needed. We found that of the three the aggressive merge is the best, reducing the space overhead (interior nodes and leaves) to a minimum. The asymptotic number of leaves in a dE-tree using the aggressive merge algorithm is about $n(n-1)/2$, which is typically much smaller than the number of leaves in a dB tree.

- Timing Study. We performed timing experiments on our implementation to gather some idea on the system response times for search and insert operations. We found that with 6 processors the system response time was about 40 milliseconds. Using table 5.10 we account for the timings we have obtained

- Analytical Performance Model We used the characteristics of the large scale dB-tree to develop a simple analytical performance model. We then studied the effect of increasing the number of processors, and found the overhead of maintaining the dB tree grows very slowly We applied the performance model to analyze the results obtained from our experiments on the dB tree with 8 processors and 50 processors With both analyses we found that the model predicts a slightly larger response time than what we obtained by our experiments Our experiments provided a response time of 50 milliseconds (figure 5 58) while the model predicted 56 milliseconds Finally, from the model the observation was that the overhead of maintaining a dB tree is not significantly affected by the node fanout as long as the fanout is large We found that a distributed search structure permits a much larger throughput than a centralized index server, at the cost of a modestly increased response time.

CHAPTER 6
CONCLUSIONS

In this dissertation we have worked on distributed B trees. Our contribution to this has been the development and implementation of several algorithms for data balancing a distributed replicated B-tree. In Chapter 1 we presented the goal of our work and provided the motivation for pursuing this research. We also presented some background on distributed data structures. We selected the B tree because of its flexibility as a distributed structure.

In Chapter 2 we discussed concurrent B trees and distributed B-trees. We also presented useful applications of the distributed B tree, namely the distributed extent tree, the dE-tree and its usefulness for parallel striped file systems.

In Chapter 3 we presented the theoretical framework of the replication algorithms developed by us. We presented two approaches, fixed position copies and variable copies. We also implemented these algorithms and they are termed full replication and path replication for the purposes of our implementation.

Chapter 4 presents the details of our implementation, the underlying architecture and details on the node migration mechanism that is fundamental to data balancing. We also presented the negotiation protocol that is inherent in our data balancing algorithms. Other details, like node structure, node naming and updates are also discussed. We have also studied the portability of our implementation by porting it to the KSR, a shared memory multiprocessor systems with 96 processors.

Finally, in Chapter 5 we presented all the algorithms that we have developed, for replication and data balancing We discussed the algorithms and their performance in detail

The performance results of the replication algorithms show that among the two methods of replication, path replication performs better and is suitable for scaling to large trees It incurs much less overhead than full replication The width of replication does not increase linearly with the number of processors and hence is suitable for scaling to a large number of processors So we took this approach and performed simulations for data balancing on large distributed B-trees

We developed centralized and distributed algorithms for data balancing and observed that distributed algorithms with sequential probing perform very well compared to the others in terms of the number of probes and moves. On an average a node moves only about 5 times in the entire system All the data balancing algorithms achieve a good data balance while incurring very little overhead We also presented the results of two different scenarios of incremental growth data balancing and fixed height tree data balancing The incremental growth performance shows patterns similar to the generalized algorithms with the width of replication being around 1.7 and the number of hops around 2.4 for 50 processors We performed experiments for fixed height trees of 3, 4 and 5 with fanout varying from 10 to 40 and the number of processors varying from 10 to 50 We observed that the width of replication varies with the number of processors only, while quickly reaching a plateau with increasing fanout The fixed height dB tree experiments show that our algorithms are suitable for larger trees with a large fanout

We simulated the distributed extent tree, the dE-tree and performed data balancing on it We developed three algorithms for balancing, namely, random, merge and aggressive merge Of these the aggressive merge algorithm does the best in achieving a good data balance with negligible overhead The asymptotic number of leaves in a dE-tree using the aggressive merge algorithm is about $n(n-1)/2$, which is typically much smaller than the number of leaves in a dB tree With the merge algorithm, we studied various extensions by changing some input parameters to the dE-tree and noted that the dE-tree performance was greatly affected by the increment size that is used to add storage to a processor when it runs short

In order to determine how well our implementation works, we performed timing experiments and studied the response times of our system With 8 processors we obtained a response time of 80 milliseconds. We have also provided an explanation of the timings we obtained (table 5 10)

Lastly we presented an analytical model to validate our experimental results We applied the analytical model to analyse our experimental results and found that the model predicts a more pessimistic time than the timing we obtained

REFERENCES

[1] Ahuja, S Carriero N and Gelernter, D , *Linda and Friends*, Computer, August 1986, Vol 19 No 8 pp 26 34

[2] Bal, H E , and Tanenbaum A S , *Distributed Programming with Shared Data* Proceedings of IEEE International Conference on Computer Languages, Miami Beach, FL, USA October 1988, pp 82-90

[3] Bal, H. E , and Tanenbaum, A S , *Distributed Programming with Shared Data*, Computer Languages 1991, Vol 16 No 2, pp 129-146

[4] Bal, H E , Kaashoek, M F and Tanenbaum, A S *Orca A Language for Parallel Programming of Distributed Systems*, IEEE Transactions on Software Engineering 1992 Vol 18, No 3 pp 190-205

[5] Baeza Yates R , *Expected Behavior of B^+-trees Under Random Inserts*, Acta Informatica 1989 Vol 26 No 5 pp 439-471

[6] Bastani, F B Iyengar, S S and I Ling Yen , *Concurrent Maintenance of Data Structures in a Distributed Environment* The Computer Journal, 1988 Vol 31, No 2, pp 165–174

[7] Bayer, R. and McCreight, E , *Concurrency of Operations on B-trees*, Acta Informatica 1972, Vol 1, pp 173 189

[8] Bernstein, P A., Hadzilacos, V and Goodman, N , *Concurrency Control and Recovery in Database Systems* Addison-Wesley Publishing Company, 1987

[9] Biliris A , *Operation Specific Locking in B trees* Symposium on the Principles of Database Systems, ACM SIGACT SIGART SIGMOD 1987, pp 159-168

[10] Carriero, N., Gelernter D , Mattson, T J and Sherman A H , *Linda Alternative to Message passing Systems*, Parallel Computing, 1994, Vol 20, No 4 pp 633-655

[11] Chang, C C and Chen C Y *A Note on Allocating k ary Multiple Key Hashing Files Among Multiple Disks*, Information Sciences 1991 Vol 55 No 1 3, pp 69-76

[12] Colbrook, A Brewer E A Dellarocas C N and Weihl W E , *An Algorithm for Concurrent Search Trees*, Proceedings of the 20th International Conference on Parallel Processing 1991 pp 38-41

[13] Dietzfelbinger, M. *How to Distribute a Dictionary in a Complete Network*, Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, Baltimore MD, USA, May 1990, pp 117 127

[14] Ellis, C S *Distributed Data Structures A Case Study*, IEEE Transactions on Computers 1985, Vol. c-34, No 12, pp 1178-1185

[15] Fan, Z and Cheng, K *A Generalized Simultaneous Access Dictionary Machine*, IEEE Transactions on Parallel and Distributed Systems 1991, Vol 2 No 2, pp 149-156

[16] Haddad, E *Optimal Allocation of Shared Data Over Distributed Memory Hierarchies*, Proceedings of the 6th International Parallel Processing Symposium, Beverly Hills CA, USA, March 1992 pp 518-526

[17] Herlihy, M., *A Methodology for Implementing Highly Concurrent Data Structures*, Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seattle WA USA, March 1990 pp 197 206

[18] Herlihy M and Wing, J , *Linearizability A Correctness Condition for Concurrent Objects*, ACM Transactions on Programming Languages and Systems, 1990 Vol 12, No 3 pp 463 492

[19] Herlihy M *Hybrid Concurrency Control for Abstract Data Types* Journal of Computer System Sciences 1991, Vol 43, No 1, pp 25 61

[20] Herlihy, M *A Methodology for Implementing Highly Concurrent Data Objects* ACM Transactions on Programming Languages and Systems, 1993 Vol 15 No 5 pp 745-770

[21] Johnson T and Shasha D , *Utilization of B trees with Inserts, Deletes and Modifies*, Proceedings of the 8th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems Philadelphia PA, USA, March 1989 pp 235 246

[22] Johnson, T and Shasha, D , *A Framework for the Performance Analysis of Concurrent B tree Algorithms*, Proceedings of the 9th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems Nashville, TN, USA, April 1990 pp 273 287

[23] Johnson, T and Colbrook, A , *A Distributed Data-Balanced Dictionary Based on the B-link Tree*, Proceedings of the 6th International Parallel Processing Symposium Beverly Hills, CA, USA, March 1992, pp 319 324

[24] Johnson, T , Krishna, P and Colbrook, A , *Distributed Indices for Accessing Distributed Data*, Proceedings of the 12th IEEE Symposium on Mass Storage Systems Monterey, CA USA April 1993, pp 199-207

[25] Johnson T and Colbrook, A. *A Distributed Replicated Data-balanced Search Structure*, 1992.

[26] Johnson T and Krishna P *Lazy Updates for Distributed Search Structures*, Proceedings of the 1993 ACM SIGMOD, International Conference on Management of Data, Washington, Dc, USA, May 1993, pp 337 346

[27] Johnson, T., *Supporting Insertions and Deletions in Striped Parallel Filesystems* Proceedings of the 7th International Parallel Processing Symposium, Newport, CA, USA, April 1993, pp. 425-433

[28] Johnson T. and Shasha D. *The Performance of Concurrent Data Structure Algorithms* ACM Transactions on Database Systems, 1993 Vol. 18, No. 1, pp. 51-101

[29] Jul E., Levy H., Hutchinson N. and Black, A., *Fine Grained Mobility in the Emerald System*, ACM Transactions on Computer Systems 1988, Vol. 6, No. 1, pp. 109-133

[30] Krishna P. and Johnson T., *Implementing distributed search structures* Technical Report TR94-009 University of Florida, 1994

[31] *KSR1 Principles of Operation*, Kendall Research Corporation Copyright Publication, 1991

[32] Koelbel C. and Mehrotra, P., *Supporting Shared Data Structures on Distributed Memory Architectures*, Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seattle, Washington, USA March 1990 pp. 177-186

[33] Kotz, D. and Ellis C. S., *Practical Prefetching Techniques for Parallel File Systems*, First International Conference on Parallel and Distributed Information Systems, Miami, FL, USA, 1991, pp. 182-189

[34] Ladin, R. and Liskov B., *Lazy Replication: Exploiting the Semantics of Distributed Services* Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing Quebec City, Quebec, Canada, 1990. pp. 43-57

[35] Boykin J. and Langerman, A., *Mach/4.3BSD: A Conservative Approach to Parallelization*, Computer Systems Journal. 1990. Vol. 3. No. 1, pp. 69-99

[36] Lee, P., Chen, Y. and Holdman, J. M., *DRISP: A Versatile Scheme For Distributed Fault-Tolerant Queues*, IEEE 11th International Conference on Distributed Computing Systems, Arlington, Texas, USA, May 1991, pp. 600-606

[37] Lehman, P. L. and Yao B. S., *Efficient Locking for Concurrent Operations on B-trees* ACM Transactions on Database Systems 1981, Vol. 6, No. 4, pp. 650-670

[38] Leiserson, C. E., *Systolic Priority Queues*, Proceedings of the Caltech Conference, VLSI, 1979, pp. 200-214

[39] Levy, E. and Silberschatz A., *Distributed File Systems* ACM Computing Surveys December 1990 Vol. 22, No. 4, pp. 321-373

[40] Litwin, W. A., Roussopoulos, N., Levy, G. and Hong, W. *Trie Hashing with Controlled Load*, IEEE Transactions on Software Engineering, 1991 Vol. 17, No. 7 pp. 678-691

[41] Litwin W, Neimat M, and Schneider D A, *Lh* - Linear Hashing for Distributed Files* Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, Dc, USA, May 1993, pp 327 336

[42] Litwin W, Neimat M, and Schneider, D A, *Rp* - A Family of Order Preserving Scalable Distributed Data Structures*, Proceedings of the 20th VLDB Conference 1994, pp. 342-353

[43] Matsliach, G and Shmueli, O, *An Efficient Method for Distributing Search Structures*, Proceedings of the First International Conference on Parallel and Distributed Information Systems Miami Beach, FL USA, December 1991 pp 159-168

[44] Miller R. and Snyder L, *Multiple Access to B-trees* Proceedings of the 1978 Conference on Information Sciences and Systems, Johns Hopkins University Baltimore March 1978, pp 400 408

[45] Parker J D, *A Concurrent Search Structure*, Journal of Parallel and Distributed Computing 1989, Vol 7, No. 2 pp 256 278

[46] Peleg, D, *Distributed Data Structures A Complexity Oriented View*, Proceedings of the Fourth International workshop on Distributed Algorithms, Bari, Italy, September 1990, pp 71 89

[47] Rao, N V and Kumar, V, *Concurrent Access of Priority Queues*, IEEE transactions on Computers, December 1988 Vol 37, No 12, pp 1657 1665

[48] Rosenberg, A L A and Snyder, L, *Time and Space Optimality in B-trees*, ACM Transactions on Database Systems Vol 6, No. 1 March 1981, pp 174-183

[49] Sagiv, Y, *Cncurrent Operations on B*-trees with Overtaking*, Journal of Computer and System Sciences, 1986 Vol 33, No 2, pp 275-296

[50] Salzberg, B *Grid File Concurrency*, Information Systems, 1986, Vol 11, No 3, pp 235 244.

[51] Salem, K and Garcia Molina, H *Disk Striping*, International Conference on Data Engineering, Los Angeles CA, USA, February 1986 pp 336-342

[52] Samadi, B, *B-trees in a System with Multiple Users*, Information Processing Letters, 1976, Vol 5, No 4 pp 107 112

[53] Seeger, B and Larson P *Multi Disk B-trees* Proceedings of the 1991 ACM SIGMOD, pages 436-445, 1991

[54] Severance, C and Pramanik, S *Distributed Linear Hashing for Main Memory Databases*, International Conference on Parallel Processing, 1990, pp 92-95

[55] Severance C Pramanik, S and Wolberg P *Distributed Linear Hashing and Parallel Projection in Main Memory Databases* 16th International Conference on Very Large Data Bases Brisbane Queensland, Australia, August 1990, pp 674 682

[56] Shasha D and Goodman N , *Concurrent Search Structure Algorithms*, ACM Transactions on Database Systems, 1988 Vol 13, No. 1, pp 53 90

[57] Tang J and Natarajan, N *A Scheme for Increasing Availability in Partitioned Replicated Databases*, Information Sciences 1991, Vol 53, No 1 2 pp 1 34

[58] Vingralek R , Breitbart Y , and Weikum G *Distributed File Organization with Scalable Cost/Performance*, Proceedings of the 1994 ACM SIGMOD, International Conference on Management of Data, Dallas, TX, USA, December 1993, pp 253 264

[59] Weihl, W E , *Commutativity-Based Concurrency Control for Abstract Data Types*, IEEE Transactions on Computers, December 1988 Vol 37 No 12, pp 1488-1505

[60] Weihl W E and Wang P *Multi-Version Memory Software Cache Management for Concurrent B-trees*, Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing 1990 pp 650-655

[61] Yen I L and Bastani F *Hash Table in Massively Parallel Systems*, Proceedings of the 6th International Parallel Processing Symposium Beverly Hills, CA USA March 1992, pp 660 664

BIOGRAPHICAL SKETCH

Padmashree, born in Hyderabad, India, is the second daughter of Mr. Achanta Apparao and Mrs. Meenakshi. She has an elder sister, Lakshmi, a younger brother Ravi and a younger sister, Rajashree.

Her schooling was at Loreto Convent, Ranchi, a hill station in Bihar, India and her basic college education at St. Xaviers College, also at Ranchi. She then obtained a Master of Science in Physics from Central University of Hyderabad. She then proceeded to do her Master of Technology in Computer Science at the Indian Institute of Technology, Madras, India.

She was then recruited as a Technical Officer in Electronics Corporation Of India, Ltd., Hyderabad. She worked there for five years on Computer Graphics, Artificial Intelligence and Parallel Processing. She then decided to further her knowledge and resigned from the organization to pursue her Ph.D at the University of Florida.

Her current research interests include networks, databases and distributed systems.

Her hobbies include aerobics, jogging, walking and handicrafts.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy

Theodore J. Johnson, Chairman
Assistant Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy

Richard E. Newman Wolfe
Assistant Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy

Randy Chow
Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy

Sartaj Sahni
Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy

Paul Avery
Associate Professor of Physics

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy

May 1995

Winfred M. Phillips
Dean, College of Engineering

Karen A. Holbrook
Dean, Graduate School